REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 15, 1998	3. REPORT TYPE AND DATES COVERED Final Report, October 8, 1997 – June 8, 1998

4. TITLE AND SUBTITLE IMPACT: Integrated Multi-Level Performance Framework for Scalable Systems	5. FUNDING NUMBER
6. AUTHOR(S) Jay Martin and Rajive Bagrodia	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Scalable System Solutions 10495 Colina Way Los Angeles, CA 90077	8. PERFORMING ORGANIZATION REPORT NUMBER SSS-9802
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency	10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	19980622 160
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The current state of art in performance technology does not facilitate detailed performance evaluation of complex, large scale systems. The aim of the Phase I effort was to demonstrate the feasibility of an Integrated Multi-Level Performance Framework for Complex Systems (IMPACT) for simulation of a parallel database system. As described in this report, the Phase I effort resulted in the design and an initial implementation of an object-oriented scalable simulator. A prototype model to simulate a parallel database machine was developed and the model was used to simulate the performance of a database running on a shared nothing architecture. The model was developed using three different simulators: CSIM, a commercial sequential simulator, PARSEC, an existing C-based parallel discrete event simulator, and COMPOSE, a new C++ library for parallel discrete-event simulation. The parallel models were *validated* against the functionally equivalent CSIM model and were found to produce identical outputs. The performance of the parallel models was subsequently studied for different workloads and substantial performance improvements were observed from parallel execution of the model. Based on the positive results obtained from the Phase I study, we have initiated development of a state of the art performance modeling tool for scalable data base systems using the COMPOSE parallel object-oriented simulator.

14. SUBJECT TERMS Parallel simulation, data base models, simulation framework			15. NUMBER OF PAGES 60
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LMITATION OF ABSTRACT

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)

IMPACT- Integrated Multi-Level Performance Framework for Scalable Systems

June 8, 1998

Sponsored by

Defense Advanced Research Projects Agency
Information Technology Office (ITO)

Issued by U.S. Army Aviation and Missile Command Under

Contract No. DAAH01-98-CR-010

Program Manager:
Dr. Frederica Darema
DARPA ITO

Technical Monitor:
Mr. Alexander Roach
US Army Missile & Aviation Command

Prepared By
Jay Martin and Rajive Bagrodia
Scalable System Solutions
10495 Colina Way
Los Angeles, CA 90077
(310) 475 1919
rlb@scalable-solutions.com

Table of Contents

1. Executive Summary 5
Primary Results 5
2. Requirements Analysis 7
2.1 Heterogeneity Requirements 7
2.2 Scalability Requirements 8
2.3 Extensibility Requirements 9
3. State of the Art in Performance Tools for Scalable Systems 10
3.1 Measurement Tools 10
3.2 Analytical Modeling Tools 12
3.3 Simulation Tools 12
3.4 Parallel Simulation 14
3.5 Parallel Simulation Tools 16
3.6 Multi-Level Models 17
4. IMPACT Simulators 19
4.1 PARSEC Simulator 19
4.2 COMPOSE 21
4.3 Complete Compose Example 28
4.4 Runtime Systems: Design Issues 29
4.5 Runtime System: Implementation Issues 30
4.6 Algorithm-Specific COMPOSE Constructs 40
5. Database Models 43
5.1 Hardware Architectures 43
5.2 Parallelism in Relational Databases 43
5.3 Data Partitioning 43
5.4 Prototype Database Models 44
5.5 Model Parallelization 46
6. Results 48
6.1 Model 1: Sequential Performance 48
6.2 Model 1: Conservative Performance 49
6.3 Model 2: Conservative Performance 51
6.4 Model 2: Optimistic Performance 52
7. Conclusion 55
8. Bibliography 57

List of Figures

Figure 1: Simulation Environment Components 30
Figure 2: Schematic of A Parallel Database 45
Figure 3 Comparative Simulator Performance on a Single Node of the Sparc 1000 48
Figure 4 Parallel Performance of a Scan query: Sparc 1000 50
Figure 5 Parallel Performance of a Query Involving Data Redistribution: Sparc 1000 51
Figure 6: Data Redistribution Query with Delay Model: Sparc 1000 51
Figure 7: Parallel Performance on 4-way SMP with Intel Pentiums and Windows NT 51
Figure 8: Speedup variation as a function of Communication Latency of target hardware 51
Figure 9: Parallel Performance of Model 2 52
Figure 10: Impact of state saving and GVT computation frequency on parallel performance 53
Figure 11: Optimistic Performance of Model 2 53

1. Executive Summary

The goal of the Phase I effort as indicated in the proposal was to
...demonstrate the feasibility of an Integrated Multi-Level Performance Framework for Complex Systems (IMPACT) by doing a detailed design of the proposed framework and complete a prototype implementation of selected components for a limited performance study of a real world parallel database system using industry standard benchmarks.......
As discussed in this report, each of the preceding objectives has been accomplished.

Primary Results

The primary accomplishments of Phase I include:

- requirements analysis of the performance evaluation environment
- a survey of the current state of the art in tools and technologies for scalable simulation environments,
- a detailed design of a scalable multi-level performance framework,
- implementation of the primary components of the framework on sequential and parallel platforms,
- design of a prototype model to simulate a parallel database machine,
- use of the model to simulate a configuration that is architecturally similar to the NCR Teradata database system.
- an experimental study on the performance of the database simulator to evaluate the potential benefits that may be derived from parallel execution of the model on shared memory parallel architectures using a variety of synchronization algorithms.
- performance comparison of the proposed simulator with existing commercial simulators

The prototype database model developed during Phase I simulates a database running on a parallel shared-nothing architecture. The hardware architecture can be viewed as a set of N nodes linked by a multi-path interconnection network such that no memory or other resources within a node is directly sharable or accessible by another node. Each node is a symmetric multiprocessor and consists of P processors, C disk controllers and D disks, where controller c_i controls d_i disks. The processors are connected to the disk controllers via a common bus. The model simulated each of the preceding components using FIFO servers; no attempt was made to capture the algorithmic details of the disk read and write operations. The model was developed using three different simulators: CSIM, a commercial sequential simulator, PARSEC, an existing C-based parallel discrete event simulator, and COMPOSE, a new C++ library for parallel discrete-event simulation.

The PARSEC and COMPOSE models were developed by SSS personnel, whereas the functionally equivalent CSIM model was developed by NCR. The PARSEC and COMPOSE models were *validated* against the functionally equivalent CSIM model and were found to produce identical outputs. The performance of the PARSEC and COMPOSE models was subsequently compared against the CSIM model for different workloads. The workload was provided by NCR as representative of two extreme types of queries: the first example represents a *scan* query on a partitioned table and contains lots of inherent parallelism. At the other extreme was a *hash-join* query that required numerous data redistributions which implies a lot of communication and possibly blocking leading to degraded performance from the parallel database. The validated simulation model was also executed on multiple parallel architectures using different parallel simulation algorithms.

The study showed that the sequential performance of the Maisie/PARSEC model was almost identical to that of the CSIM model; however the performance of the COMPOSE model implemented on a single node was almost three times better than the sequential CSIM model! Further, for the scan query, the performance of the parallel COMPOSE model was faster by a factor of 5.7 compared to the sequential COMPOSE model and almost 17 times faster compared to the sequential CSIM model! The speedup for the hash-join query varied with the hardware configuration of the target system, but significant performance benefits were demonstrated for both sequential and parallel models.

Based on the encouraging results obtained from the Phase I study, we have initiated development of a state of the art performance modeling tool for scalable data base systems using the COMPOSE parallel object-oriented simulator

2. Requirements Analysis

The first step in the design of the performance environment was to complete a requirements analysis for the proposed system. The requirements were divided into three primary categories: *heterogeneity, scalability,* and *extensibility.*

2.1 Heterogeneity Requirements

The set of requirements covered in this section refer to the ability of the performance framework to model different systems at varying levels of detail possibly using different modeling paradigms.

Support for end-end performance models Performance frameworks that target only software, hardware, or application level models are unlikely to yield significant insights into the performance characteristics of applications. While low level models can be extremely useful in designing individual components (e.g. transistor level models for the design of VLSI circuits), system designers are primarily interested in studying the impact of specific design alternatives on the performance of the application. This means that the model must be developed in layers, where each layer models the lower layers at an appropriate level of abstraction, but the workload that is used to drive the model can be transformed to derive appropriate responses from the lower layer models. Considering a database system as an example, IMPACT must be capable of developing models of the hardware system that consists of a set of processors, each with its own local memory and disks and the interconnection network that links the processors; model the representation of the database as it is distributed among the multiple disks and processor memories and the architecture of the query processing software system; and finally model the workload as a traffic generator.

Support for multi-level, hierarchical models As systems become arbitrarily large, detailed simulation models for such systems may be computationally intractable even with parallel execution of the models. It should be possible to model each subsystem at multiple levels of granularity, from coarse to detailed, and compose a model of the system that includes subsystem models that have varying levels of granularity. For instance, it may be desirable to model each node of a parallel machine as a single delay server when evaluating the impact of different decomposition strategies for a given query in a database system. On the other hand, while evaluating the impact of different declustering or partitioning strategies for the relations in a database across multiple disks, the node must be modeled at a greater level of detail.

Hierarchical modeling of subsystems allows a given layer in a end-end performance model to view the lower layer as an abstraction, while maintaining the required degree of detail at each layer in the system model. Hierarchical modeling also allows the model itself to be designed in a top-down iterative manner that imposes established software engineering discipline on the model design process.

Support for hybrid models A hybrid model is a partially implemented design, where some components exist as simulation or analytic models and others as operational subsystems realized in hardware or in software [Bagrodia & Shen 1991]. Hybrid models directly support integration of measurement and modeling frameworks. This paradigm allows an analyst to ascertain the impact of design changes in one subsystem without developing detailed simulation models of the entire system.

Hybrid models have a number of advantages over simulation models, particularly for on-line monitoring. For domains like network protocols or parallel software, operational software is typically faster than a detailed simulation model; integrating operational subsystems in the model can lead to better model execution times. Second, every simulation model is an abstraction of a system and is hence only an approximate representation of the corresponding subsystem. Inclusion of operational subsystems typically implies an improvement in the overall accuracy of the model. Third, hybrid models are particularly useful when evaluating complex systems, parts of which have already been implemented. Rather than design simulation models of the existing subsystems, the operational system may itself be integrated into the model.

Seamless integration of measurement & modeling in a common framework A coordinated measurement, modeling, and simulation approach is essential to evaluate complex systems. Measurements on operational systems allow the identification of the critical subsystems that are the first order performance bottlenecks. Models allow us to investigate the impact of alternative implementations for the bottlenecks, and to identify future bottlenecks that might emerge in the redesigned system. An integrated measurement and modeling approach is thus iterative in nature; it ensures that the performance of the system can be monitored continuously during its lifetime such that existing performance problems are correctly solved and future performance problems can be anticipated before they develop into critical bottlenecks.

2.2 Scalability Requirements

The performance framework must be scalable along numerous dimensions: it must support the design of large and complex models, it must support the evaluation of very large systems within a reasonable duration of time, and it must serve as a conduit to eventual system deployment.

Reduce design costs of complex systems: The primary motivation for a performance framework must be to enable system designers to tradeoff design alternatives prior to the prototyping stage, so that components in deployed systems are not found to be sub-optimal, requiring expensive and time consuming redesign and implementations. The success of IMPACT depends on its ability to reduce the design cost of a real world system.

Ability to predict performance of very large systems The scalability of the performance framework is relevant from two perspectives: first, IMPACT must be capable of specifying arbitrarily large configurations. Second, IMPACT must be capable of executing models for arbitrarily large physical configurations. In other words, if the physical system is scaled up by a factor of *s*, it is desirable if the response time of the model does not increase by more than a factor of *s*. Alternately, it should be possible to use additional resources (scaled by a factor no larger than *s*) to maintain the same response time for the model. Parallel execution offers an attractive potential to scale the modeling framework.

IMPACT supports multiple simulation protocols and diverse parallel architectures to execute parallel simulation models. It must be able to simulate database models with many millions of relations stored on tens of thousands of disks connected to thousands of processors.

2.3 Extensibility Requirements

Both the measurement and modeling components of IMPACT must be extensible. It must permit different types and granularities of measurements to be specified for a given system and experiment. The application level measurement framework must be portable such that if the application is ported among different systems, the instrumentation can also be ported with minimal changes. This would greatly facilitate a comparative evaluation of the performance of a given application as a function of different hardware and software system parameters ranging from processor configurations to interconnection network characteristics and the input-output system configuration.

Support for migration of simulation models to prototypes The costs of model design and maintenance has been another drawback to widespread use of advanced performance technology. The design and development costs for detailed simulation models for complex systems can easily rival corresponding costs for the physical systems themselves.

To preserve the significant investment in developing detailed simulation models, it is desirable to support the transition of the simulation models into operational software. Not only does such a port allow effective reuse of code, it also ensures consistency between the model and the eventual implementation. IMPACT must support a methodology and tools for the migration of simulation models to software prototypes.

Support for visual, interactive interface for model design A visual and interactive model design interface can help to rapidly configure a system from existing components, build alternative models from these components and built-in entity templates in a simple visual framework, specify the metrics that are of interest to the analyst, interactively debug the simulation and monitor its execution, changing the level and detail at which the performance metrics are being collected, and optimize these models for parallel execution, if necessary.

The Visual interface may be used to design new models, as well as to integrate existing libraries into a model. Modular design allows large structures to be built in ever larger increments. A set of connected entities may be grouped together as a module (e.g. a node that consists of a processor and disks). The module may be replicated and also nested in higher level modules which supports the design of a hierarchy of models with an arbitrary level of nesting.

3. State of the Art in Performance Tools for Scalable Systems

Large scale performance studies have used a variety of tools and techniques ranging from measurements, analytical models, simulation models, and hybrid techniques that use a combination of the preceding approaches.

3.1 Measurement Tools

Consistent and accurate measurement techniques are useful in understanding the performance of operational systems and in identifying the primary bottlenecks in their execution. This is particularly true of large parallel and distributed systems, where the sheer number of individual components and the complexity of their interactions make it hard to collect data that is time synchronized across the system. Although a large number of measurement packages are available ranging from simple profilers to sophisticated instrumentation and analysis frameworks, we only examine systems that have been successfully used to measure a diverse set of parallel applications and have been ported to multiple parallel architectures.

We survey the state of measurement technology in three primary areas that are relevant to this project: parallel applications running on massively parallel architectures, networked systems, and parallel database systems. Example measurement systems in the first category include the Pablo [Reed et al 92] measurement suite from University of Illinois which has been used to monitor, analyze, animate, and display the performance of parallel programs on shared memory and distributed memory architectures and Paradyne [Miller et al 95] from University of Wisconsin that can be used to add instrumentation to object code without the need for recompilation. In the area of network measurements, frameworks like SNMP (Simple Network Management Protocol) and the Universal Measurement Architecture (UMA) have been commonly used. In the commercial database arena, all vendors provide customized performance monitoring capabilities, but relatively few systems are used widely across diverse platforms and vendors.

Parallel Programs

Three primary techniques have been used to measure the performance of software systems: *profiling*, *sampling*, and *event traces*[Reed et al 92]. Profiling is perhaps the simplest and best known technique that is commonly and transparently supported by most operating systems. A profiler allows a programmer to identify the application modules that consume the greatest percentage of the systems resources (typically expressed in terms of cpu cycles, main memory, and disk accesses). However for parallel applications, profiling can produce a lot of disconnected and unrelated data that is hard to interpret. Profiling also provides summary data over the entire execution of the program rather than as a time varying stream. Sampling can be used to provide useful performance information in a temporal context. However, the sampling frequency is typically not related to specific system states and there is little correlation between the sampled values and the system state. The sampled data does not provide adequate insight in to the performance of the system.

Event traces are an abstraction that allow a programmer to relate all activities in the system, to the extent they can be instrumented, that are spawned by a specified event. What constitutes an 'event' is specified by the programmer using the constructs supported by the instrumentation framework. All system wide activities, ranging from local computation, message traffic, and disk accesses triggered by the event maybe collected to produce a complete and time ordered sequence of relevant activities. The primary drawback with this is that unless used wisely, it tends to produce a fire hose of data that must first be processed and reduced automatically to permit meaningful analysis.

One of the well-known trace-based instrumentation system is Pablo [Reed et al 92], a portable and extensible instrumentation library that has been used to characterize the performance of a variety of applications on parallel and networked computer systems. Pablo provides a graphical interface to specify the necessary instrumentation on a given parallel program. The instrumented program is then compiled and linked with a trace capture library that actually records the specific trace information in a standard format called SDDF for self-describing data format. SDDF data may then be analyzed and displayed using a variety of data visualization tools. The trace information collected by the library can be specified at three different levels of detail ranging from summary counts that simply count the number of occurrences of an event (e.g. entry of a given procedure) to time intervals that measure the total time that elapses during the execution of a given logical code fragment.

Network Measurement Systems

SNMP [Stallings 96] was designed to be an application-level protocol that is part of the TCP/IP protocol suite. The proposed architecture uses a manager-agent model, where the agents reside on the nodes of the managed network and their data collection activities are supervised by the manager. The manager periodically polls the agent to obtain collected measurements. Alternately the agents may generate *trap* messages to send data to the manager. A management information base (*MIB*) is used to collect and organize the statistical data that is collected by the framework. A RDBMS MIB has also been defined to collect generic and vendor-specific data when SNMP is used to monitor a database system.

UMA is similar to SNMP in that it also uses a manager-agent monitoring model, with the primary difference that UMA supports a per-peer or agent-agent communication model. Also, unlike SNMP which uses a connection-less UDP based communications to exchange performance metrics, UMA uses TCP/IP. Performance information is typically sent autonomously by the agents to the manager. UMA uses a hierarchical data format with implicit timestamps. Unlike SNMP, which defines a relatively small set of metrics, UMA has been designed to update a very large set of performance metrics (e.g., 160 different fields for UNIX and another 160 for databases like Oracle). Although there is substantial overlap between SNMP and UMA, it appears that whereas he former is more suitable as the network management protocol, the latter provides a more comprehensive and easily extensible solution for system performance management [Gunther 98].

Database Systems

Much of the work in the commercial marketplace in the area of database performance evaluation is based on measurements. Relational Data Base Management Systems (RDBMS) sometimes use transaction monitors to aid load balancing for OnLine Transaction Processing (OLTP). Commonly used monitors include Tuxedo and Encina [Gunter 98]. A number of vendors offer sophisticated toolkits to tune the performance of operational databases. For instance Precise Software Solutions offer the Precise/SQL system to monitor and tune the performance of Oracle databases[Precise]. Menlo Software offer the DBAware [Menlo] tool to monitor and tune the performance of Oracle and Sybase databases. A number of other vendors offer similar suite of tools that allow users and database administrators to measure the performance of operational databases while they are processing queries to identify patterns of resource usage. However these tools do not provide significant help either in predicting the scalability of these systems as both the databases and the hardware systems get larger. Neither do they provide any assistance for capacity planning or evaluation of alternative data management strategies (e.g., different data partitioning techniques).

3.2 Analytical Modeling Tools

A large number of analytical techniques, and tools that exploit these solution methods, have been developed [Allen 94]. To a large extent, the popularity of the analysis tools is their ability to estimate critical performance metrics like response time, throughput, and system utilization using a few simple properties of the systems. Typically, the system is described using a network of queues, where each queue is characterized by properties that include inter-arrival time of customers, service time distributions for jobs, number of servers, service discipline of the queue, system capacity, and routing probabilities for departing jobs. The most commonly used technique for the solution of queuing networks (circuit of queues) is Mean Value Analysis or MVA. Of course, MVA is known to be applicable in only a subset of situations, where the queuing network satisfies the restrictions due to the so called product form networks [Jain 91]. As the computations for the MVA algorithm are recursive, for sufficiently large N, the computation may become computationally intractable. A number of approximations have been suggested to reduce the computation time at the expense of some loss of accuracy, but the primary problem with analytical techniques is the limited applicability of the technique. A large number of system characteristics have been identified, which if present in the system, will render it unsuitable (or at least make the solution considerably harder) for analytical models. These characteristics include the presence of blocking at one queue that may inhibit processing at another queue, bulk or load-dependent arrivals, mutual exclusion, contention, queue defections, and non-exponential service times [Gunther 98]. Analytical techniques have been used successfully to evaluate limited facets of computer system performance, including large data base systems.

3.3 Simulation Tools

Measurement requires that the system being measured be deployed and accessible to instrumentation and minor perturbation. While this is normally possible for operational systems, this approach is not applicable for large scale systems that are in the process of being designed. Further, it is typically infeasible to modify characteristic parameters of an operational system across a wide parameter space to study the impact of the changes on the actual system performance. Analytical techniques are useful for rapidly evaluating a set of alternative solutions to identify the most promising alternatives. However, their primary drawback is their limited applicability. The complex interactions among the various components of scalable systems like wireless networks and database systems means that it is only possible to construct approximate analytical models of the system and detailed analysis of the system via analytical models is not feasible. Simulation is the most commonly used performance evaluation technique for such systems. A number of commercial tools are available both as general purpose simulators (e.g. Simscript, Modsim, SESworkbench, CSIM, etc.) as well as domain specific tools (e.g., BoNeS, COMNET (CACI), CPT (Rooftop Communications) and OPNET (Mil3) for simulation of network protocols). However, most of them use only sequential model execution restricting their utility for large models and none of them provide a path towards eventual implementation of the protocols. University simulators include NS from Lawrence Livermore Labs[McCanne et al], Morpheus [Abbott 92] from University of Arizona, NetSimPro [Short 95] from UCLA, and the interactive network simulator from Xerox Parc.

A number of other domain-specific simulators have also been developed including parallel program simulators like the Wisconsin Wind Tunnel[Reinhart et al 93], LAPSE[Dickens et al 96] and MPI-SIM [Prakash 96]; system software simulators like SimOS[Rosenblum 95] and SESAME [Bagrodia et al 97]; and parallel architecture simulators like Tango [Davies wt al 91] and Proteus[Brewer et al 91] and many others. Most simulators for database systems are constructed from general purpose simulators like those developed to evaluate the Gamma[Dewitt et al 90] and Bubba[Boral et al 90] data base systems. Commercial data base vendors tend to develop specialized in house simulators or use general purpose simulation tools for limited evaluations (e.g., the TCP tool from NCR described in the previous section). The one exception to this appears to be the recent tool from SES for the simulation of relational databases.

Although simulations are widely used for performance studies, a number of factors have limited their use in evaluating scalable, heterogeneous systems:

- **Resource intensive model design**: In many simulators (e.g., OPNET, SES Workbench, etc.) no direct path is provided from the simulation model to system prototypes. This requires that the system designer begin the coding and implementation phase from scratch after a proposed design has been simulated to resolve performance issues which increases product development costs and hinders future investments in performance models.

- **Model Scalability**: Execution times for sequential simulation models have become a significant bottleneck in effective use of these models. For instance, slowdown factors of 30-50 per processor are typical in the simulation of parallel programs. This implies that simulation of a query that executes for 5 to 15 seconds on 128 nodes of a contemporary multi-processor architecture can take anywhere from a few days to many weeks on state of the art sequential workstations! As the size of the physical system is increased, the memory requirements for the model can easily outgrow the memory capacities of sequential workstations.

- **Heterogeneous models**: For end-to-end performance analysis, it is necessary to model systems at all levels ranging from applications, software, and hardware. Typically, subsystems at different levels are required to be modeled at different levels of detail, and may involve the use of different modeling paradigms that must be supported in an integrated manner.

Some recent efforts in developing advanced performance technology for scalable systems have produced simulation environments that address the preceding concerns. In particular, the PARSEC and COMPOSE simulators developed at UCLA and Scalable System Solutions respectively. These tools share the following primary characteristics that address the preceding problems: First, they provide an easy path for the migration of simulation models to operational software prototypes. Second, they are among the few simulation environments that has been implemented on both distributed and shared memory platforms, and which supports a diverse set of parallel simulation protocols. Third, they directly integrate parallel and hybrid model execution techniques within the overall framework of system simulation. Lack of such integration has been an important hindrance to widespread utilization of parallel simulation.

3.4 Parallel Simulation

A simulation consists of a series of events, which must be executed in the order of the time stamps placed on them. On a single processor, these events can be placed in a central queue to ensure their correct ordering (the Global Event List algorithm). When run in parallel, however, the event list is distributed such that each processor has only a portion, and events may arrive asynchronously from other processors. Thus, additional information is required to ensure that each processor executes events in their correct order. To this end, three primary types of parallel synchronization protocols have been described in the literature: conservative[Misra 86], optimistic[Jefferson 85], and mixed[Jha & Bagrodia 95], where the last may include sub-models that execute in either conservative or optimistic modes.

Parallel simulation models are commonly programmed as a collection of logical processes (or LPs), where each LP models one or more physical process in the system. Events in the physical system are modeled by message communication among the corresponding LP; each message carries a time stamp that represents the time at which the corresponding event occurs in the physical system. Henceforth, we will use the terms event and message interchangeably. The following variables are defined for each LP[Jha & Bagrodia 95]:

- Earliest Output Time (EOT): The EOT of an LP is a lower bound on the time stamp of any future messages that may be sent by the LP. If EOTs is infinity for some LP s, the remaining LP can be executed independently of s. A sink process is an example of such an LP.

- Earliest Input Time (EIT): The EIT of an LP is a lower bound on the time stamp of any future message that may be received by the LP. the EITs of an LP s is the earliest time that it may receive a message from another LP. The EIT of an LP is infinity if no other LP send messages to it; a source process is an example of such an LP.

- Lookahead: The lookahead for an LP is the future time interval over which the LP can completely predict the events which it will generate. The lookahead is used to calculate EOT. For instance, consider a FIFO server that serves each incoming job for delta time units. If the server is idle at some simulation time t, its lookahead is delta and its EOT is (t+delta).

In its most commonly used forms, a PDES model is either optimistic (all LP executed in the optimistic mode) or conservative (all LP executed in a conservative mode). A conservative LP cannot tolerate causality errors (events executing out of timestamp order); hence it will only process events with timestamps less than its EIT. A number of algorithms have been designed to compute the EIT of each LP in a distributed manner. Many of these have been implemented in PARSEC and COMPOSE. In general, the lookahead and communication topology of a model have a significant impact on the performance of conservative algorithms. The communication topology of a model is described by maintaining *predecessor*- and *successor*-sets at each LP, that respectively refer to the set of LPs from which an entity may receive messages or to which it may send messages.

An optimistic LP may process events with timestamps greater than its EIT; however, the underlying synchronization protocol must detect and correct violations of the causality constraint. The simplest mechanism for this is to require an optimistic LP to periodically save (or checkpoint) its state. Subsequently, if it is discovered that the LP processed messages in an incorrect order, it can be rolled back to an appropriate checkpointed state, following which the events are processed in their correct order. An optimistic algorithm is also required to periodically compute a lower bound on the timestamp of the earliest global event, also called the Global Virtual Time or GVT; checkpoints timestamped earlier than GVT can be reclaimed. Using our model, it is sufficient for an optimistic LP to preserve at least one checkpointed state with a timestamp smaller than its EIT. (The minimum of the EIT of all optimistic LP is a reasonable lower bound on the GVT of the model).

Given appropriate mechanisms to advance the EIT and EOT of the conservative or optimistic LP, it is possible to implement a PDES model which is composed from optimistic & conservative sub-models. In general, any of the GVT computation algorithms, conservative algorithms, or even a combination of the preceding algorithms can be used by a PDES to compute the EIT of each LP, regardless of the execution mode of the individual LP in the model. The choice of a specific algorithm for a given scenario is an efficiency rather than a correctness issue. We outline an aggressive null message based scheme: whenever the EOT of an LP, say s, changes, EOTs is sent using a null message to other LP; the null message may of course be piggy backed on a regular message when feasible. On receipt of a null message, an LP recomputes its EIT and EOT and propagates changes to other LP. It is easy to show that given a model with no zero delay cycles (i.e. a cycle of LP all of which have a zero lookahead), such an algorithm will eventually advance the EIT (and hence the time) of every LP, regardless of whether it executes in conservative or optimistic mode. The hybrid protocols are also useful for the composition of autonomous simulators, where each simulation may internally use a conservative, optimistic, or sequential protocol.

3.5 Parallel Simulation Tools

Available tools for large scale parallel simulation range from operating systems dedicated to support of parallel simulations to domain-specific packages. The primary advantage of the OS-based approach is that it provides complete flexibility in the choice of a programming language, and that much of the scheduling, memory management, and IPC functionality provided by the OS need not be duplicated by the parallel simulator. The major drawback of this approach is the sheer complexity of implementing an operating system while tracking the multiple parameters that eventually determine its performance. A second drawback is that at least in some implementations, each simulation object is an OS thread and is treated as a heavy-weight process; such an implementation is efficient only if the computation contains coarse grain parallelism and can be efficiently decomposed to exploit this characteristic. Operating systems that have been designed to support parallel simulations include the Time Warp Operating Systems from JPL and the MIMDIX system from Georgia Tech.

At the next level of abstraction are scalable simulation languages (PSLs) and libraries that provide programmers with a set of well-defined constructs to design (parallel) models. PSLs provide a set of model definition primitives together with a set of parallel programming primitives for process (or thread) definition, creation and interprocess communication and synchronization. The primary advantage of PSLs over the OS approach is that the former can provide programming primitives that can be supported efficiently by the underlying simulators and eschew those that are hard to implement. Secondly, a well-designed language can present the simulator to the programmer at an appropriate level of abstraction. Transparency of the underlying implementation is desirable from a software engineering viewpoint but may lead to inefficiencies in model implementations. The primary disadvantage of PSLs is the need to learn new languages, although most PSLs are designed by extending a familiar base language with a small set of primitives.

At the end of the spectrum are domain-specific tools that are designed to serve a specific application area like logic simulation of VLSI circuits. The obvious advantage of this approach is that the simulator can exploit specific application characteristics to reduce overheads of parallel model execution and present an application-specific interface to the programmer to facilitate model description. Three areas that have shown significant potential for performance improvements include circuit simulations (with tools like PLDVISIM from UNC, MIRSIM from UCLA, and PVHDL from Cincinnati among others), parallel program models (tools like LAPSE from NASA, Wisconsin Wind Tunnel from Wisconsin, and MPI-SIM from UCLA), and communication network models (with tools like GloMoSim from UCLA, SimKIT from Calgary). In each of the preceding areas, performance improvements of at least one order of magnitude have been obtained for realistic applications. Our goal is to develop an efficient, scalable simulator for parallel database systems using a general purpose PDES environment.

Common approaches to design of general purpose parallel simulation software include:

- Library based approaches represented by systems like GTW, UPS, COMPOSE, and SPEEDES. These systems typically provide simulation and parallelism capabilities via calls to libraries implemented in standard sequential languages like C or C++. Their primary advantage is that the user does not have to learn a new language. A major drawback is that because no translator is used, the library routines must provide a less abstract interface than is typically possible with a high-level simulation language. Some systems have targeted a specific domain. For instance, UPS proposes an interesting extension to existing sequential simulators by exploiting parallelism in restricted ways with specific simulation objects like queues that are known to have good lookahead properties.

- Enhancement of sequential (simulation) languages with primitives for parallel simulation; examples include Sim++ [Baezner et al. 1990], APOSTLE [Wonnacott and Bruce, 1996], Maisie [Bagrodia and Liao 1994], MOOSE, an object-oriented extension of Maisie, and PARSEC, a next generation implementation of Maisie. The ability to use a translator allows languages to provide a more succinct and 'natural' interface for the programmer as compared with simulation libraries. It is also possible to implement a number of optimizations like automatic granularity control as in APOSTLE and automatic reduction of rollback distances as in Maisie.

In this project we used a parallel simulation language (PARSEC) and a parallel simulation library (COMPOSE) to develop parallel simulation models of parallel database systems.

3.6 Multi-Level Models

A multi-level modeling methodology allows initial abstract or coarse models of a system to be iteratively refined into detailed models for critical subsystems. The coarse simulation model is used to predict behavior at very large scales, where detailed models may be intractable even with parallel model execution. Initial coarse models also identify the subsystems that are performance bottlenecks and require detailed representation.

The refined simulation model captures the implementation details of the critical subsystem. Assume that the interconnection network was found to be the primary performance bottleneck in a coarse model of the database. The simple contention free model used in the coarse model may then be replaced by a switch-level model that models the queuing and contention at each intermediate point of the network to identify the primary cause of the congestion. The modeling framework must allow the detailed network model to be inserted with minimal modifications to the rest of the model. In the best case, the other components are not modified at all and the interface compatibility issues are addressed by using adaptive and intelligent interface code.

The inclusion of detailed model characteristics allows the refined models to estimate performance measures not derivable in the coarse grain simulator (e.g., message loss due to congestion). Once these parameters have been estimated in the fine-grain model, they are subsequently introduced as macroscopic parameters in the coarse grain simulator to evaluate larger systems. Thus, an iterative modeling approach is used where each subsystem is eventually modeled at the level of detail necessary for the performance study.

The hierarchical approach is well suited to the interactive design and implementation of parallel systems. It permits us to carry out conceptual algorithm design on large topologies in an efficient, interactive manner using the coarse grain simulator. In this high level design process, the implementation details are hidden behind a few key parameters which were derived from the fine grain simulator. In turn, the high level designs tested in the coarse simulator are implemented in the fine grain simulator. Eventually, the fine grain simulator will serve as a conduit to software implementation as described in the previous section. This design and evaluation cycle is closed by feeding back the measurement results to the simulation model to further improve its accuracy for predicting the performance of large-scale models.

Support for multi-level models requires the design of a rich library of models that include subsystem models at various granularities and/or the ability of a language to support adaptive interfaces between model components that allow a coarse model of a subsystem to be replaced by a detailed model, without requiring significant changes in other component models. In Phase I, we have developed low fidelity and approximate models for each of the subsystems for parallel databases. In Phase 2, we expect to develop detailed models of the critical subsystems and develop capabilities to incorporate multi-level models.

4. IMPACT Simulators

Two different simulators have been used to develop the database models described in this report: Maisie together with its newer incarnation PARSEC, and COMPOSE. We present a brief description of PARSEC and a complete description of the COMPOSE simulation environment.

4.1 PARSEC Simulator

PARSEC, a Parallel Simulation Environment for Complex systems is a parallel simulation language that uses the process-interaction approach to discrete event simulation. PARSEC has been implemented on both distributed and shared memory platforms, and supports both conservative and optimistic parallel simulation protocols. Supported conservative protocols include the null message based schemes as well as protocols based on the conditional event algorithm, whose performance has not been studied widely. The optimistic protocol is based on the space-time paradigm, an early implementation of which is described in [Bagrodia et al 1991]. PARSEC also supports the Ideal Simulation Protocol or ISP which can be used by an analyst to compute a tight lower bound on the execution time of a parallel simulation model on a given architecture [Bagrodia et al 1998]. Although there are other tools that have been implemented on multiple architectures or support multiple simulation protocols, to the best of our knowledge, no existing simulation environment provides such a broad coverage of parallel architectures and protocols. PARSEC has been used to simulate a diverse set of applications including VLSI circuits, parallel programs, parallel IO systems, interconnection networks, electronic LANs, ATM networks, aeronautical communication networks, and wireless communication protocols.

A PARSEC program consists of a set of modules, where each module is an entity or C function. Each entity is an LP that models a corresponding physical process. Entities can be created and destroyed dynamically and multiple entities can be mapped to a single processor. For parallel execution, each entity is explicitly mapped to a processor when it is created and cannot subsequently be migrated to a different processor. Events are modeled by message communications among the corresponding entities. Each entity is associated with a unique message buffer and asynchronous send and receive primitives are provided to deposit and remove messages from the buffer. An entity in a PARSEC model simulates the local actions of a physical object using C code and describes the interactions of the corresponding physical object via timestamped messages. It supports both the hold and wait until primitives introduced earlier. The PARSEC implementation of a wait until statement allows many complex enabling conditions to be expressed directly, without requiring the programmer to describe the buffering explicitly. The resume condition can be a complex expression and may reference (though not modify) state variables of the entity and the message buffer. We present two examples to illustrate PARSEC constructs for parallel simulation.

The first example illustrates the asynchronous message passing constructs used by PARSEC using an entity called *manager* to simulate a resource manager. The first three lines declare the types of messages that may be sent or received by the entity: type *request* that is used by another entity (with id *his_id*) to request a certain number of units (*num*), type *release* to return resources to the pool, and type *done* to inform the requesting unit that the requested number of units are available. The body of the entity is basically an infinite loop with a receive statement to process the two message types (*request* and *release*) that may be received by the entity. We note that the request message includes a Boolean expression or *guard*. The guard has two expressions: the first ensures that a given request is accepted by the entity only if the required number of units are currently available with the manager. The second expression ensures that if the buffer contains both *request* and *release* messages, *release* messages are accepted by the manager first.

```
message Request {int num; ename his_id;};
message Release {int num;};
message Done {};
entity Manager (int maxa)
{       int units=maxa;
        for (;;)
        receive   (Request   req)   when   ((req.num<=units)   &&
        qempty(Release))
        {       units -= req.num;
                send Done to req.his_ id;
        }
        or receive (Release rel)
                units += rel.num;
}
```

Resource Manager

The next code fragment illustrates the simulation constructs supported by PARSEC. The fragment simulates the transmission of a data packet by an entity to another entity with id *next-hop*. The first statement sends a message of type *data-packet*. The next statement causes the entity to suspend until one of three events occur: it receives an *ack* message (line 2); it receives a *nack* message (line 5); or it receives a timeout message (line 8). The timeout message is a conditional message; the entity will receive his message only if it does not receive an ack or a nack message within the specified time-out interval -- *round-trip-delay-time*. If it receives an ack message, the message is copied into local variable a; the entity executes a hold statement to simulate the time required to process the message and executes function *process_ack* to simulate the actions executed by the corresponding physical process. A *nack* message is processed similarly. The definition of the various message types have been omitted for brevity.

Every PARSEC program must include an entity called *driver*. Execution of a PARSEC program is initiated by executing the first statement in the body of entity driver. As mentioned earlier, PARSEC supports a number of parallel simulation protocols and architectures. The specific protocol and architecture to be used in a given execution of a model is specified as a command line option. A complete description of the language is available in the Parsec User Manual [UCLA 1998], and a report on the parallel performance of the simulator with a variety of applications has been summarized in [Bagrodia et al 1998].

```
 1 send data-packet{data} to next-hop;
 2 receive {      (ack a)
     {
3    hold(ack-time);        /* suspend for time to process an ack  */
4    process_ack_received(a);          /*  in the physical system  */
     }
     or (nack n) {
6     hold(ack-time);               /* suspend for time to process a nack  */
7     process_nack_received(n);                  /*  in the physical system  */
    }
8   or timeout after (round-trip-delay-time) {
9     resend_data_packet(data);
    }
  }
```

Data Transmitter

4.2 COMPOSE

COMPOSE (Conservative, Optimistic and Mixed Parallel Object-oriented Simulation Environment) is a C++ library for executing parallel discrete-event simulations on both shared and distributed memory parallel computers. The environment is a pure C++ class library design and does not require special preprocessing or compilation. COMPOSE models simulation entities with C++ objects that send time-stamped messages which correspond to simulation events. To ensure correct simulation event execution order, the runtime implements sequential and parallel simulation algorithms.

COMPOSE uses the process-interaction approach to discrete event simulation, which assumes that a parallel simulation is composed of a set of simulation objects (or entities) that do not share state and communicate exclusively by time-stamped asynchronous message passing. When a message is sent to another entity, it corresponds to scheduling an event at that other entity.

COMPOSE uses the "concurrent object model" for message processing: when a message is processed, a method of the object is executed to completion. This can also be looked at as an asynchronous method call on the object with the message as the method's (input) parameter. This model is different than the usual process model that has explicit receive statements and a thread of control. To provide the object model (without using compilers or preprocessors), the runtime requires that the simulation programmer explicitly register at runtime the proper "simulation action" method for each received message type. These action methods perform changes to the simulation entities and send messages to other entities. COMPOSE messages are user defined C++ structs that inherit from class "MessageType".

4.2.1 Why COMPOSE

In Phase I of this study, we investigated the feasibility of using both PARSEC and COMPOSE for the proposed database simulator. The following factors influenced our decision to use COMPOSE to develop the proposed simulator: *first*, while PARSEC is based on C, COMPOSE was designed using C++ and all the benefits of object-oriented programming are directly available to the simulationist. In particular, multi-level models using associative broadcasts are more effectively implemented in an object-oriented environment, where inheritance and virtual function concepts may be used to provide the appropriate interfaces in a modular manner. *Second*, unlike PARSEC which is a language, COMPOSE is a library; programmers familiar with C++ do not have to learn a new programming environment and hence may be more inclined to use this system. For instance, existing commercial C++ program development environments (e.g., Visual C++) can directly be used to develop COMPOSE programs. *Third*, the sequential performance of the database models in COMPOSE was almost twice as fast as PARSEC. The primary reason for the superior performance of the COMPOSE is that it significantly reduced the number of distinct simulation objects at run time, which increases computational granularity and reduces context-switching overheads. *Last*, the simulation runtime system for COMPOSE is designed to promote interoperability and we intend to explore the use of existing object frameworks like DCOM and CORBA both to support object distribution within COMPOSE and for integration of COMPOSE with other tools.

4.2.2 COMPOSE Constructs

COMPOSE provides various routines for performing simulation initialization, entity creation, unconditional event scheduling (message passing) and conditional event scheduling (timeouts). These runtime capabilities are provided by the "EntityType" base class. The programmer creates an entity class by inheriting from "EntityType" and implementing the data structures and action methods. The action methods perform the events and call methods from the "EntityType" base class (for example, "SendMessage"). Once all entity classes have been defined, the programmer writes a "Starter Entity". The "Starter Entity" is a special entity that is used to initialize the simulation. Its job is to create all simulation entities and to seed the simulation with initial messages. After it does its job, the "Starter Entity" is destroyed and the simulation is executed. The simulation then executes until there is no activity in the simulation or the specified end simulation time has been reached.

The primary constructs in the design of a simulation model in COMPOSE include simulation entities, messages, and timeout events. We briefly describe these facilities in this section.

Messages

Messages are the communication mechanism used by COMPOSE. Entities send time-stamped messages to other entities and these messages cause simulation events. Messages are implemented as simple C++ structs that are derived from a base class called

"MessageType". The "MessageType" base class provides the message header needed by the COMPOSE runtime to process the message. It includes such things as the message's time-stamp and the source and destination of the message, but these variables are declared private and cannot be accessed by the simulation programmer.

Example Message Type Declaration:

```
class UserMessageType: public MessageType {
public:
   // User-specified parameters.
   int  Parm1;
   char Parm2;
};
```

To work with distributed memory parallel machines, COMPOSE messages must be completely self-contained and thus must not contain data-structures with pointers. With shared memory machines, pointers are possible but message data-structures must not be shared and the message type must have a virtual destructor to reclaim the dynamic memory. Messages must also be allocated from the heap with "new". To minimize overhead, COMPOSE does not copy messages and overrides "new" for message types.

Simulation Entities

In COMPOSE, objects in the simulation (henceforth referred to as "entities") are represented as C++ objects. For this purpose, the COMPOSE library provides a base entity class called "EntityType" which provides a set of useful member functions (runtime operations) and encapsulates information necessary for the simulation environment. Because we are executing a possibly distributed simulation that has simulation time ordering, entities cannot call each others method functions directly. Instead, messages are sent and queued in the runtime for each entity and in the proper time order, the messages are given to the entity. COMPOSE has a concurrent object model for message processing: a message is processed by the entity by executing a method of the C++ object. These methods are executed to completion and there is no capability of context switching from a method execution. These methods have one parameter that corresponds to the specific message type it will process. A runtime binding mechanism is used to associate a message type with its corresponding method. The library provides the routines "RegisterInitMethod" and "RegisterMethod" to specify this message type to method mapping. These binding calls must be made by the simulation programmer in the object's constructor.

Example entity class:

```
#include "compose.h"
class UserEntityType: public EntityType {
public:
   COMPOSE_ENTITY_ENVIRONMENT(UserEntityType);
   // Initialization Message definition.
   class InitMessageType: public MessageType {
   public:
      // Initialization Variables.
      ...
   };
private:
   // Define entity methods.
   void InitMethod(const InitMessageType& InitMessage);
   void JobMethod(const JobMessageType& JobMessage);

   // Define constructor with method registration calls.
   UserEntityType() {
      RegisterInitMethod(InitMethod);
      RegisterMethod(JobMethod);
   }

   // State Variables
   ...
};//UserEntityType//
```

Message Type to Method Binding Runtime Calls

The following runtime call associates a user method with a message type so when a message of that type is processed by an entity the corresponding method is called. These calls should occur in the entity's constructor. If a message type is not registered by an entity, receipt of a messag eof that type will cause a runtime error. This makes COMPOSE a dynamically typed simulation object model.

- **REGISTER_METHOD(UserMessageType, ActionMethodPointer);**

This runtime call registers the "ActionMethodPointer" with the runtime and binds it with the specified "UserMessageType". An entity may selectively receive certain messages by specifying guards as shown below

- **REGISTER_METHOD_AND_GUARD(UserMessageType, ActionMethodPointer, GuardMethodPointer);**

When the guard method, specified by the **GuardMethodPointer** returns "False", the entity can not process messages of this type and the message is deferred. The message can be processed in the future when this method evaluates to "True".

Initializing and Creating Entities

Once an entity is defined, it can be created and initialized with method calls in this section. In COMPOSE, creation and initialization are done in two steps. First the entity is created causing the entity's constructor to be executed, then the entity is initialized by sending an initialization message to the just created entity. This causes the method mapped to initialization message type to be executed. These actions are done synchronously which means the runtime waits until these actions complete before it returns from these methods. This eliminates race condition problems at simulation startup at the cost of decreased parallelism during the entity creation.

- **CREATE_ENTITY(EntityType, NodeID, NewEntityName) ;**
- **CreateEntity(&EntityType::COMPOSE_New, NodeID, NewEntityName);**

This method creates an entity of type "EntityType" on the virtual node "NodeID" number. Virtual node Ids are "int"s and go from 0 to (NumberOfNodes - 1). IDs outside this range will be "mod-ed" to be within this range. The name of the created entity is returned (out reference parameter) in "NewEntityName" (of "EntityNameType"). This returned name is used in calls such as "SendMessage" to identify the entity

The explanation for the non-macro version requires a digression about how creation is implemented in a distributed memory (but homogenous) environment. To create an object on another node, the constructor for the specific entity type needs to be called on that node (in a different address space). The way this is done in COMPOSE is by using a pointer to the entity's constructor. The first problem is that C++ doesn't provide a pointer to a constructor feature. The simple and standard workaround for this deficiency is to define a trivial function that just calls the constructor and use a pointer to the trivial function. This trivial constructor function is defined in the "COMPOSE_ENTITY_ENVIRONMENT" macro and is called "COMPOSE_New()". This is why the CreateEntity method must have "&EntityType::COMPOSE_New" as its first parameter. The macro version simply hides this ugliness. Given the runtime has a pointer to the "COMPOSE_New" function, another problem is that this pointer is from the sender's address space and probably cannot be used in the destination node address space. COMPOSE uses the fact that the execution binaries on the two nodes are identical and tweaks the passed pointer to point to the right executable code.

- **InitEntity(EntityName, InitMessagePointer);**
- **INIT_ENTITY(EntityName, InitMessagePointer);**

This runtime call Initializes the entity with name "EntityName" (of EntityNameType) with the Message with pointer "InitMessagePointer". Note that the message must be allocated with "new". Example of an entity creation and initialization:

```
CREATE_ENTITY(MyEntityType, NodeID, NewEntityName);
INIT_ENTITY(NewEntityName, new MyEntityType::InitMessageType(...);
```

The reason that creation and initialization are done separately is that most simulation topologies are cyclic and thus neighboring entity names needed for the entity's initialization may not yet exist. The usual method to build a simulation is simply to create all the entities and then initialize them with entity name of their neighbors.

Sending Messages

An entity can send messages to other entities in the simulation with the following methods:

- **SendMessage(EntityName, MessagePointer);**

This method sends a message pointed to by "MessagePointer" (of type "MessageType*") to the entity identified by "EntityName" ("EntityNameType"). The message must be allocated with "new" (Passing an address of a stack or global variable will cause an error). Once a message has been sent, it becomes the property of the receiving entity and must not be modified or reclaimed by the sending entity. Passing a pointer eliminates being forced to copy the message. A message with a delyaedtime stamp is sent by the following method:

- **SendDelayedMessage(EntityName, MessagePointer, DelayTime);**

Timeout Events

The basic idea behind the "timeout event" is for an entity to wait for events for a specified length of time and if no messages come in then to "timeout" and do some special processing. If a message does come in, then the timeout event is cancelled. The following method allows the simulation programmer to setup the timeout:

- **ScheduleTimeout(TimeoutMethodPointer, TimeoutTime);**

The action taken by the timeout is passed in the first parameter as a pointer to parameter-less method ("TimeoutMethodPointer"). The time that the timeout will occur is the second parameter "TimeoutTime" (of type "TimeType")). One prickly issue with timeout events is defining when the timeout occurs in relation to events with the same timestamp. For example, if a timeout event is scheduled for time 10 and there is a regular event scheduled at time 10, does the event cancel the timeout? The default in COMPOSE is for the timeout to occur and to be the first event for a particular timestamp. This can be overridden at the entity level so that timeouts only execute if they will be the last event for that timestamp:

Executing timeouts as the last event has problems with zero delay cycles. In the simplest case, suppose the timeout event sends a message to itself with zero delay, then the "executed last" property has just been violated. This will actually cause an infinite rollback sequence in the optimistic parallel algorithm. Executing timeouts last also effects the conservative parallel protocol as it puts stricter requirements on lookaheads. It is highly likely that the conservative protocol will require a positive delay when sending a message from a timeout event.

Building and Starting Simulations

COMPOSE can be executed in a shared memory environment or distributed memory environment. Execution of the COMPOSE model begins with the exeuction of main(); for shared memory implementations, "main()" is executed and the simulation system starts up parallel threads for other simulation objects. In a distributed memory environment, N copies of "main()" are executed in parallel, one on each node. This causes a problem as normally the programmer only requires a single one simulation routine to initiate execution. COMPOSE gets around this problem by requiring the simulation programmer to make a special temporary object called the "Starter Entity" that is executed only on one node (node 0) in a distributed environment.

The Starter Entity's job is to create and initialize the simulation entities (with "CreateEntity" and "InitEntity") and seed the simulation with initial messages (with "SendMessage"). After it does this it is killed and the simulation starts executing. The Starter Entity's simulation construction code can be in either its constructor or its initialization method.

The following calls are provided by the library to build and initiate execution of the model. For the first option, only the constructor is executed:

- **Start_COMPOSE(&StarterEntityType::COMPOSE_New, NumberVirtualProcesses);**
- **START_COMPOSE_NO_INIT(StarterEntityType, NumberVirtualProcesses);**

In the next option, both constructor and initialization methods are executed:

- **Start_COMPOSE(&StarterEntityType::COMPOSE_New, InitMessage, NumberVirtualProcesses);**
- **START_COMPOSE(StarterEntityType, InitMessage, NumberVirtualProcesses);**

These routines create the "Starter Entity" and then execute the simulation. There are two versions, one that requires an initialization message ("InitMessage" of type MessageType) and one that doesn't (entity initialization is skipped). Note that the initialization message for Start_COMPOSE does not have any of the restrictions on pointers and references as normal messages and can viewed as just a regular object that the programmer is passing into the system. The first parameter, "&EntityType::COMPOSE_New" is the pointer to the starter entity's constructor (see create entity command section). "NumberVirtualProcessors" (of type int) is the number of threads to create in a shared memory environment (ignored in a distributed architecture). The macro versions simply hides the "::COMPOSE_New" syntax.

Example main program:

```
int main() {
     Start_COMPOSE(&StarterEntityType::COMPOSE_New, 4);
     return 0;
}
```

Entity And Simulation Termination

Entities are deleted by the runtime immediately after they execute "TerminateMe()" in a method and at the end of the simulation. When an entity is deleted, its C++ destructor is called. In COMPOSE, the destructor code is designated to be "outside the simulation" and must not effect the state of the simulation itself. This means that messages cannot be sent from within a destructor. A destructor could write some simulation results to a file or store them in some global data-structure.

A simulation completes if there are no more events to process or the simulation has reached the designated "end of simulation time" (see "SetEndOfSimulationTime"). When the simulation completes, for every existing entity, the destructor is executed and the entity is deleted. (This is the reason why destructors are not allowed to contain code that sends messages). After all simulation resources have been cleaned up, the call to Start_COMPOSE completes.

Starter Entity Runtime Methods

These runtime calls are defined in class "EntityType" and can only be called at the beginning of the simulation in the Starter Entity. The purpose of these runtime routines is to define the global properties and algorithms to be used by the simulation.

- **SetEndOfSimulationTime(const TimeType& EndTime);**

  Sets the end of simulation time. Events after the time "EndTime" will not execute.

- **DoNullMessageSynchronization();**

  Execute using the null message algorithm.

- **Set_GVT_ProcessingIntervalRealSeconds(IntervalTime);**

  Set the real interval time between GVT calculations. "IntervalTime" is a float. The primary use of this routine is to tune how often GVT is calculated when running the optimistic algorithm. If it is calculated more frequently, then the amount of memory used state saving will be reduced at the cost of greater GVT calculation overhead.

4.3 Complete Compose Example

The following is the complete C++ source for a simple FIFO queue server entity. The server simply accepts jobs (messages) in FCFS order and forwards them to another entity after delaying them by a duration that corresponds to its service time. The InitMethod sets up the communication topology for the network for use by conservative and adaptive algorithms. In this example, the server sends messages it processes back to itself. The end of simulation time is set so the simulation will terminate.

```
const ServiceTime = 10;
class JobMessageType: public MessageType {};

class ServerEntityType: public EntityType {
public:
   COMPOSE_ENTITY_ENVIRONMENT(ServerEntityType);

   class InitMessageType: public BaseMessageType {
   public:
      EntityIDType SuccessorEntity;
      InitMessageType(EntityNameType SuccEntity) :
         SuccessorEntity(SuccEntity) {}
   };
private:
   // Define entity methods.
   void InitMethod(const InitMessageType& InitMessage);
   void JobMethod(const JobMessageType& JobMessage);
   // Define constructor with BindMethod statements.
   ServerEntityType() {
      REGISTER_INIT_METHOD(InitMethod, InitMessageType);
      REGISTER_METHOD(JobMethod, JobMessageType);
   }

   // State Variables
   EntityIDType SuccessorEntity;
};//ServerEntityType//


// InitMethod initializes the entity's variables
// and provides the topology and lookahead information
// for conservative or adaptive algorithms.
```

```
void ServerEntityType::InitMethod(const InitMessageType& InitMessage)
{
   SuccessorEntity = InitMessage.SuccessorEntity;
   AddSuccessorEntity(SuccessorEntity);
   SetLookahead(ServiceTime);
}//InitMethod//

void ServerEntityType::JobMethod(const JobMessageType& JobMessage)
{
   Hold(ServiceTime);
   SendMessage(SuccessorEntity, JobMessageType);
}//JobMethod//


// Simple Starter Entity that builds the simulation.

class StarterEntityType {
public:
   COMPOSE_ENTITY_ENVIRONMENT(StaterEntityType);
Private:
   StarterEntityType() {
      SetEndOfSimulationTime(1000);
      EntityNameType EntityName;
      CREATE_ENTITY(ServerEntityType, 0, EntityName);
      INIT_ENTITY(EntityName,
         new ServerEntityType::InitMessageType(EntityName));
      SEND_MESSAGE(EntityName, new JobMessageType());
   }//StarterEntityType()
};

int main() {
   Start_COMPOSE(&StarterEntityType::COMPOSE_New, 1);
}
```

4.4 Runtime Systems: Design Issues

A portable kernel has been designed to execute COMPOSE programs on sequential and parallel architectures (distributed memory and shared memory) using both conservative and optimistic algorithms. The set of simulation algorithms currently supported supported (or anticipated to be available shortly) to be used with COMPOSE include:

- sequential or Global Event List algorithms,
- three parallel conservative algorithms that respectively use null messages, conditional events, and a combination of the preceding two schemes called the accelerated null message protocol (ANP),
- a parallel optimistic algorithm based on space-time simulations, and
- the Ideal Simulation Protocol (or ISP) which is based on the concept of the critical path to predict a realistic lower bound on the execution time of a given parallel model.

The kernel provides a unified simulation runtime system to implement the preceding set of synchronization algorithms on a variety of architectures as shown in Figure 2. For a given experiment, the programmer specifies the synchronization algorithm to be used as a command line option, which causes the appropriate library to be linked in with the runtime system. The preceding set of algorithms are currently available under PARSEC. In some cases, COMPOSE shares the same code base, and in others the code has been re-implemented when it is possible to exploit additional efficiencies given the simpler interface that has been defined for COMPOSE.

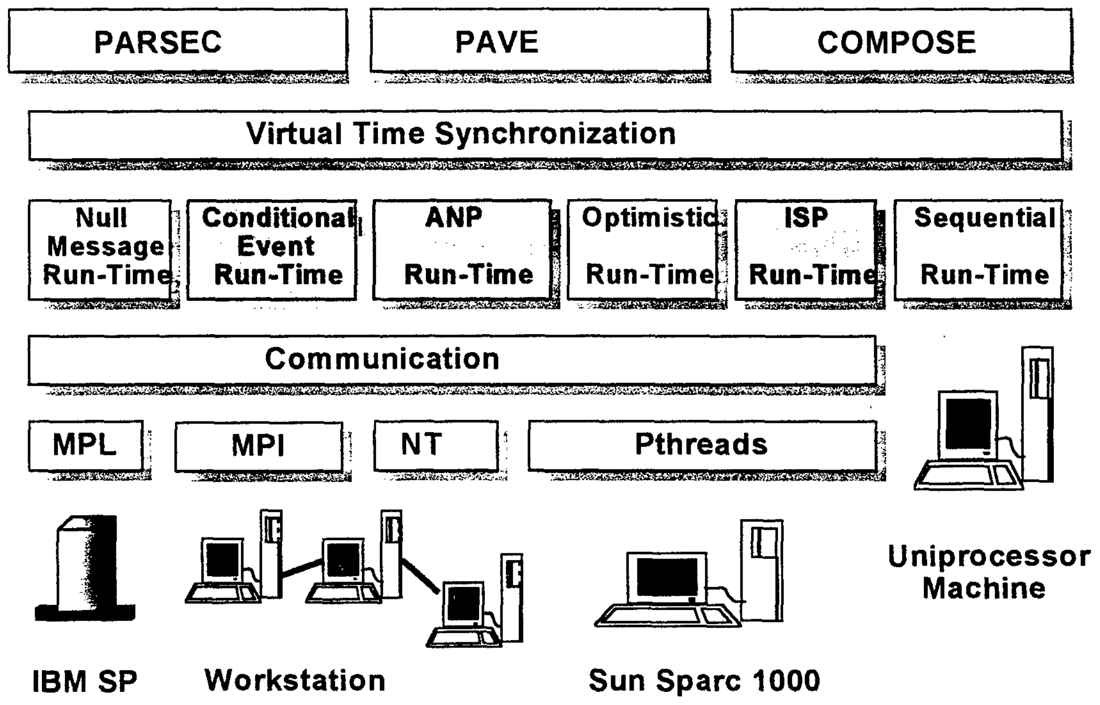

Figure 1: Simulation Environment Components

The ISP algorithm allows an analyst to estimate a realistic lower bound on the execution time of a parallel simulation program. ISP computes the execution time for a given model on a given architecture by executing the model using a complete message trace for that model. Using the trace, each entity can locally determine the order in which incoming messages are to be processed, without using any synchronization protocol. Consider the following example: an entity receives a message m, with timestamp u, when the EIT of the entity was v; v<u. In general the entity can process this message only when its EIT is u or larger. Assuming that the entity has no other messages that can be processed, it must remain idle for the time interval that its EIT remains less than u. However, using the message trace, ISP pre-computes the sequence in which incoming messages are accepted by the entity, thus allowing it to process the 'next' message in the sequence as soon as it arrives; in the preceding example, if m was the next message in its sequence, it can be processed by the entity as soon as it is received and no blocking overhead is incurred. In this manner, ISP excludes overheads that are due solely to the simulation protocol, but includes all overheads that are due to the model partitioning, message transmission and buffering, and other factors related to the execution of a parallel program. This allows an analyst to determine if the observed inefficiencies in the execution of a parallel simulation model are primarily due to implementation inefficiencies in the simulation protocol or to the inherent lack of concurrency in her parallelized model.

4.5 Runtime System: Implementation Issues

The COMPOSE runtime is constructed from the following object types and subsystems:

- Messages
- Entity Objects
- Manager Objects
- Shared Memory Communication Buffers.
- Distributed Communication System.
- Synchronization mechanisms

Entity and Message Objects

An entity object encapsulates the code and data structures defined by the user-level entity and provides the system-defined (library) data-structures necessary for execution of events in the runtime system. It includes code to manage the message queue of the entity, including adding and removing messages from the queue. This object also performs all the simulation-related actions on behalf of the corresponding user-level entity. For instance, when executed in the optimistic mode, it performs the necessary state saving, rollback, and garbage collection activities. Similarly when the object is executed in the conservative mode using the null message algorithm, necessary computation to handle incoming null messages, compute the EIT, EOT and transmit null message activities are performed by this object.

Messages are the primary means of interaction among the simulation objects. An event that occurs at some time T at a simulation object P, is implemented in the runtime system by transmitting a message with timestamp T to the simulation object that corresponds to the object P. All messages in the system are derived from the pre-defined COMPOSE class called 'MessageType'.

Entity Manager Objects

Although entities represent a single thread of control for the programmer, the run-time system defines a layer of manager objects, each of which manages a set of entities. Each manager object corresponds to a single schedulable thread that is visible to the OS. On a distributed memory platform, each processor will normally contain a single manager object. Entities that are managed by the same manager object are said to be local, whereas entities managed by different manager objects are said to be remote. The manager object is responsible for scheduling the entities it manages, as well as maintaining the appropriate state information for each entity, handling the creation and deletion operations, retrieving messages from the communication interface that are destined for one of its entities, transmitting messages to remote entities, etc. In addition the manager schedules synchronization algorithm specific activities like GVT calculations, null message sends, etc. The runtime system is designed such that communication between remote entities must go through their respective managers. On a shared memory architecture, the managers communicate through shared memory communication buffers. Because entities are accessed only by their owning manager's thread, entities are not concurrently accessed by multiple OS threads and thus locks are not necessary to maintain the integrity of an entity's data-structures.

Inter-manager Communication

On a distributed memory architecture, the communication among the manager objects is typically implemented via calls to the standard message passing libraries like MPI or by a vendor provided communication capability. On a shared memory architecture, a memory buffer is provided for each manager. Communication among managers is implemented by reading and writing the messages to the appropriate buffer. As buffers may be accesses simultaneously, appropriate locking mechanisms must be used to ensure exclusive access to the buffer. A number of optimizations have been implemented to improve the cache behavior at each manager and these are discussed subsequently.

4.5.1 Entity Scheduling

As discussed in the previous section, each entity manager corresponds to a parallel thread. The manager schedules entities in an order determined by the execution mode of the simulator. In sequential and optimistic modes, the scheduling queue is a priority queue that orders entities by the timestamp on its earliest executable event. The priority queue is implemented as a heap where the entities float up and down inside the heap. Execution of an event will cause an entity to move further down in the heap, and receiving an early message will cause an entity to move up in the heap. An embedded link back from the entity allows a constant access time to be maintained for each entity. Using this data structure causes all events to be executed in the strict order of their timestamps. This requires an O(Log N) reordering of the queue after each event.

For conservative execution, the entity manager keeps a list of all conservative entities and a simple FIFO queue of conservative entities that are ready to execute some event (i.e. it has at least on event with a timestamp smaller than its safe time). Each entity is removed from the FIFO queue, in turn, and all safe events at that entity are executed. The entity is descheduled only when it has no events to process. Note that the queue management overheads for the conservative algorithm is much less than the sequential/optimistic priority queue because queue manipulations for the non-sorted FIFO queue is a constant time operation and, unlike the case with sequential algorithms, multiple events may be executed following each dequeue operation.

4.5.2 Entity Message Queue and Event Execution

A COMPOSE entity specifies a number of message types that it may receive. During entity creation, each message type is bound to a method and an optional guard as described in section 4.2.2. An incoming message is processed by the entity by executing the corresponding method, only if the associated guard (if any) evaluates to true; other wise he message is deferred. Deferred messages are stored in a local queue at the entity until a future time when the guard evaluates to true. Because COMPOSE has the ability to defer messages by type , the message queue for an entity is implemented as a list of sub-queues, one for each message type. Each sub-queue is implemented as a splay-tree for fast O(log N) insertions and deletions. The RTTI feature of C++ is used to differentiate message types so they can be filed into the proper sub-queue. A number of optimizations have been added to improve the performance of optimistic simulations. For instance, when an entity is rolled-back, the rolled-back messages are not reinserted back into the splay-trees but are instead stored in linked lists separated by message type. Because these messages are rolled-back in sorted order, no ordering is necessary and insertion is constant time versus the O(log N) time that would be necessary to insert them into the splay-trees. This technique does require that the head of the link lists to be checked and compared to the heads of the splay-trees when retrieving the next message for processing.

A guard function can be associated with each message type. The guards must be re-evaluated following every method executed by the entity. Following the execution of a method, the runtime identifies the message types that are not deferred (referred to as active types) and determines the earliest timestamp for messages in the *active* message queues. This is the earliest event time for the entity and is used by the manager entity to make its scheduling decisions. When this event is scheduled for execution, the method that is to be executed is identified by a simple table lookup (recall that each message type is mapped to a unique local method by the simulation programmer during entity creation). The specified method is executed with the corresponding message as a parameter.

The "timeout" event is handled as a special case separate from the message queues. The timeout message may be used in one of two modes: *timeout first* and *timeout last*. Given a timeout scheduled for execution at time T, timeout first will execute the message before any other messages with that timestamp. Timeout last guarantees that it will be executed only after all messages with that timestamp have been executed. Timeout last can be used to collect all messages of a certain timestamp by scheduling a timeout of duration 0. In COMPOSE, the "timeout method" which performs the timeout processing, is registered dynamically using the "ScheduleTimeout" library call described in (4.2.2)

4.5.3 Message Communication in Shared Memory

In the shared memory implementation, each manager thread has an incoming message buffer into which other managers can deposit messages. Rather than deposit entire messages, only message pointers are actually inserted. Since these buffers are accessed concurrently, they must be locked. A simple dual buffer technique is used so that the destination manager can empty a buffer while other threads are adding more messages to the alternate buffer. The buffers are implemented as contiguous arrays so that a single cache-line will contain a good sized block of the transferred message pointers. This minimizes the number of cache lines that need to be retrieved when the buffer is emptied. Minimizing cache line movement is a critical factor for efficient execution of a parallel simulator.

In optimistic mode, these buffers also distinguish between regular messages and anti-messages so that an anti-message can also be sent by merely sending a pointer to the original message.

Message Memory Caches

Message memory is cached in COMPOSE to lower the number of calls to the system allocate and de-allocate routines, which are considerably more expensive. Each manager thread maintains its memory cache locally and first tries to obtain the memory required to transmit a message from the local pool. On receiving a message, the receiver thread gains ownership of the message and the corresponding memory. This scheme has been referred to as "sender pools" in the literature. The message pool at each entity is organized as a set of lists, where each list contains free message blocks of a given size, with the sizes organized into ranges. A default set of ranges is provided by COMPOSE, but can be overridden by an entity. Large messages outside these size ranges are not cached. The sender pool scheme can lead to imbalances in the size of the local memory pools if , for example, an entity is a message sink. To handle imbalances, COMPOSE maintains a central memory pool. Messages are transferred to this pool by a manager, if the number of messages at a manager thread gets above a certain threshold. When a manager exhausts its local pool, it checks the central pool before allocating additional memory from the OS.

4.5.4 Message Communication in Distributed Memory

In the distributed memory implementation, COMPOSE relies on MPI to implement communication among remote entities. The communication subsystem collects information needed by the distributed GVT algorithm. Specifically, it tracks the sequence numbers and the message simulation times of messages that it sends to and receives from other nodes. This information is periodically sent to the central GVT algorithm calculator and is used to compute the GVT as explained in section 4.5.8.

4.5.5 Entity Creation and Initialization

Creation in COMPOSE is complicated because of the "library only" limitation as well as the need to run in distributed memory. Basically, to create an arbitrary entity object on a remote node, the runtime systems must call the constructor of the entity object which resides in a different address space on the destination node. This requires two problems to be solved: first, C++ does not provide a pointer to a constructor feature. The simple (and standard) workaround for this deficiency is to define a trivial function that just calls the constructor and using a pointer to the trivial function. This trivial constructor function is defined in the "COMPOSE_ENTITY_ENVIRONMENT()" macro that must be inserted into the user's entity classes. The second problem is that the pointer must point to the appropriate location in the address space of the destination node. COMPOSE uses the fact that the execution binaries on the two nodes are identical and tweaks the pointer passed by the creator o point to the right executable code.
Entity creation and initialization is done synchronously which means the creator entity is blocked until these actions complete. Synchronous creation helps to prevent race conditions in message communications.

4.5.6 Null Message Algorithm

COMPOSE provides an implementation of the null message conservative algorithm. The null message algorithm works by sending null messages that give guarantees about the earliest message that could be sent from one entity to another. By taking the minimum of these guarantees, the entity can determine whether it can safely execute an event. The null message algorithm uses the message communication topology to reduce the number of null messages. This topology is represented by the predecessor- and successor-sets described in section 4.6.2 . These sets can be viewed as input and output channels from and to other entities. For each predecessor, there is an "Earliest Input Time" (EIT[i]) for the channel, the minimum of all these EITs is the overall EIT for the entity. Each channel to a successor gets an Earliest Output Time (EOT[j]) from the entity. There are two modes to calculate EOT, one mode with a single global EOT and one mode that calculates channel specific EOT[j]'s. To carry out this calculation the simulation programmer needs to specify lookaheads and "EOT_Ceiling"s. The "lookahead" is the minimum delay before sending a message after receiving a message and the EOT_Ceiling puts a limit on the EOT calculated (for supporting "Timeouts"). Depending on the mode there may be one pair of these values or one pair for each output channel. The equations for calculating EOT or EOT[j] are as follows:

- EIT = min(EIT[i]).
- EarliestEventTime = min(EIT, CurrentEntityTime, EarliestMessageInQueueTime)
- EOT = min((EarliestEventTime + LookaheadTime), EOT_Ceiling)
  or
- EOT[j] = min((EarliestEventTime + LookaheadTime[j]), EOT_Ceiling[j])

Entities propagate EOTs by sending null messages. The entity managers usually schedule sending null messages and entities do not necessarily send null messages every time an EOT changes. Entities piggyback EOTs onto regular messages on distributed memory architectures. Currently the runtime calculates EITs, EOTs and flushes null messages when there are no events to process. Another possibility would be to flush null messages after so many events.

In the conservative algorithm, the GVT value is a lower bound on the earliest event time currently in the simulation. If an EIT is less than GVT, then the EIT can be increased to the GVT. This could be used to jump the null message algorithm forward if there was a large quiet period in the simulation. Normally when running the conservative algorithm, the GVT algorithm is only used for termination, i.e. when it detects that there are no events left to be executed in the simulation.

4.5.7 Null Messages in Shared Memory

Null messages are used to notify an entity of a new "Earliest Input Time" for a channel. In shared memory, instead of making and sending null messages, the system can simply set the EIT variable directly at the destination entity without using locks. This requires moving only one cache line. This saves the overhead of sending an actual message that must be created and inserted into the concurrent message buffers (which requires using locks). This is possible because there is only one writer and only one reader. Another nice property is that Channel EITs always increase. The downside of this scheme is that the entity and its manager no longer have notification when the Channel EITs change and thus the runtime must poll these variables. This requires the entity managers to schedule when the entities EIT (minimum of all channels) will be calculated. Because the "null messages" are bypassing the communication buffer used by the regular messages, there is a possibility that the "null messages" will pass the regular messages and be processed in the wrong order. To stop this from occurring, the runtime first calculates EITs (gets the "null messages"), then it empties the normal messages in the communication buffer and finally it calculates the EOTs needed for outgoing null messages.

Currently the entity manager simply notifies all entities to perform EIT and EOT calculations and do null message sends. It does not keep lists of entities with pending calculations or null messages. This is reasonable, as the conservative algorithm normally needs a steady flow of null messages flowing around the system to be efficient so it is likely that each entity will have null messages to send. This is especially true given that the manager does null message calculations only when it has run out of executable events.

4.5.8 Optimistic Algorithms

Entity State Saving and the Event Queues

In optimistic mode, an entity's state is saved so the entity can be rolled back. This requires a set of queues to keep track of past events, states and output messages. COMPOSE supports periodic state saving which implies that the entity's state may not be saved for each event. This naturally causes a distinction between event and entity state data-structures. The past event data-structure has the following information:

- A reference to the event's message.
- The "preempt time", i.e. if there is a message before this time the event must be rolled back.
- A reference to the list of output message that the event sent. This is needed for sending anti-messages.
- An optional reference to a copy of the entity's state.

The runtime implements an event queue that records past events by inserting new event records at the tail of the queue. Rollbacks remove incorrect events from the tail of the queue and the old events are deadwood collected from the front of the queue. It is important that the event record be as small as possible as each event record is memory overhead that will slow the simulator down (cache effects) in comparison with the sequential simulator.

Each event record also keeps track of the message that corresponds to the event and all messages generated by the event. This allows these message to be cancelled via anti-messages. The event record can also have a reference to a copy of the state of the entity before the event was executed. This allows the state of the entity to be restored during rollback. Note that all data-structures used by the runtime implement memory caching so as to minimize the use of the OS memory management system calls. For example, the above queues will keep a free list of "queue cell memory blocks" for reuse. This means that these data-structures will grow to their maximal size and not shrink. Currently, there is no memory cache flushing protocol in COMPOSE.

Periodic State Saving with Coast Forward

Because of periodic state saving, the closest saved state may be earlier than the desired state. To restore the entity to the desired state, the runtime "coasts forward" from the saved state. A "coast forward" re-executes already completed events merely to restore the state of the entity; output messages generated by the entity during this phase are discarded.

Anti-message rollbacks

An entity message may be cancel a message that was sent earlier by sending a *anti-message*. When an entity receives an anti-message, it checks to see if the message has been processed. If not, the message is simply deleted from the message queue. If it has been processed, an anti-message rollback event is scheduled by the destination entity (a variable is set) and the incoming message s marked appropriately. In shared memory architectures, this marking can be done directly by the sender by maintaining a pointer to the original message. Eventually the anti-message rollback event is executed assuming it is not preempted by an even earlier anti-message or straggler message rollback. The anti-message rollback causes the entity's state to be restored to before the cancelled message's execution and this event and all later events are canceled. For each cancelled event, the event's message is put back into the message queue (if it has not been killed) and anti-messages are sent for all sent messages.

Straggler rollbacks

A *straggler* message refers to a message that is received by its destination after a message with a later timestamp has been processed. An entity schedules a straggler rollback event when it receives a straggler. Execution of this event causes the entity to be rolled back to a checkpoint preceding the timestamp of the straggler, cancel all events (and messages) that were executed and re-execute the events following the straggler. Note that the straggler event does not cancel events of the same timestamps except for "timeout last" timeouts which must be the last event for that timestamp.

Rollback with Lazy Cancellation

When an entity is rolled back, it typically cancels all events that it generated since the event that is being rolled back. As discussed earlier, cancellation is implemented by sending anti-messages. With lazy cancellation, an entity that is rolled back does not send the anti-messages immediately, but rather stores them locally in the *Lazy Cancelled Message List(LCM)*. During a subsequent re-execution, if the sequence of generated messages are a complete subsequence of messages in the LCM, they are simply discarded. If a generated message is different, the remaining messages in the LCM must be cancelled by sending the corresponding anti-messages and the new messages are sent to their destinations. Lazy cancellation may prevent unnecessary cancellation and re-execution of a given event. For example, if a straggler message simply causes the event to be queued within a destination entity, the events generated by the entity following a rollback will perhaps be identical to the ones that were generated during the preceding computation. Note that the implementation must include messages on the LCM when computing the GVT of the model.

Entity State Saving

Periodically, COMPOSE checkpoints all of an entity's state into "state objects" to support rollback and recomputations. The following information is saved:

- All time varying status variables in the runtime (the entity's time, etc).
- A memory dump of the user's entity object.
- All dynamically allocated objects from the state saved memory manager.

The runtime provides a user accessible state-saved memory allocator that keeps track of blocks of memory so that can be saved and restored when a rollback occurs.

Copy State Saving

The state of the entity is represented as a list of memory blocks. This includes both the entity object itself and dynamically allocated memory in the entity's changeable state. Note that these memory blocks cannot be moved because the user may have pointers in their data-structures that reference the addresses in these blocks. The concept behind the big block system is to provide a software layer to support different small block memory allocators which use the big blocks. The memory blocks can also have a dynamic size, i.e. the last used address can be specified so that unused memory does not have to be saved.

To save the entity state, the list of entity state memory blocks and the entity object's immediate memory are copied into a "memory block queue". The process is reversed to restore the state of the entity. Conceptually the "memory block queue" allows for the saving of memory blocks of many varied sizes in a packed sequential manner. Memory blocks are saved at the back of the queue and rollbacks will cause memory from the back of queue to be reused. Garbage collection reclaims memory from the front of the queue. Because of the properties of optimistic execution, memory will never have to be reclaimed from the center of the queue. The implementation of the "memory block queue" is a doubly linked circular list of fixed sized blocks. In this circular queue of fixed sized blocks, the "state blocks" being saved can start at any position within a fixed size block and can span any number of fixed blocks. The queue is expanded as needed and by leaving the allocated memory in the circular queue when it becomes unused, the queue serves as "memory cache" (which reduces calls to the OS memory allocator). There may need to be a periodic mechanism that flushes this cached memory back to the OS.

One complication is the addition and deletion of blocks to the entity's state block list. Previous saved states will not have the same state block set and thus copying the old block set over the new block set will not work. To handle deletion, the "state block list" keeps deleted blocks around until the states that include the block are deadwood collected. To handle insertion, the system keeps track when the block was created to detect and when the creation is rolled-back the block can be deleted.

Rollbackable Data structures

Rollbackable data structures remember past states and can roll backwards in time. Pointers to these objects are kept on a list and when a rollback or deadwood collection occurs each of these data-structures are notified. C++ allows for these objects to automatically register themselves with the runtime. This is done by placing code in the base rollbackable class's constructor to register itself with the currently executing entity. This allows for automatic registration without bothering the user.

```
class RollbackableType {
public:
   RollbackableType();
   virtual void Rollback(int EventNumber) = 0;
   virtual void void GarbageCollect(int BeforeEvent) = 0;
private
   PointerType(EntityType) EntityPtr;
};//RollbackableType//

RollbackableType::RollbackableType() {
   EntityPtr = EntityType::CurrentThreadsEntityPtr();
   EntityPtr->RegisterRollbackableObject(this);
}//RollbackableType()//
```

GVT Algorithms and Deadwood Collection

The Global Virtual Time (GVT) is a lower bound on the earliest message in the system. In optimistic simulations, events before GVT can be reclaimed ("deadwood collected") as these events can never be rolled back. When the GVT is calculated, it is immediately distributed through out the system.

Currently the runtime schedules the GVT algorithms to execute periodically every "D" real time seconds, where D is a user-specified parameter that should typically be set to a small fraction of a second. A designated thread (Thread 0) is in charge of controlling the GVT calculation along with its other normal simulation duties.

Simple shared memory GVT algorithm

The shared memory GVT calculation algorithm is a simple non-blocking algorithm based on time intervals. In this algorithm, each virtual processor thread stores the earliest timestamp on any message sent to another thread during the current interval. The beginning and end of the intervals are designated when the "Interval Local Virtual Time" (Interval LVT) is calculated.

The algorithm is as follows:

1. The thread calculating the GVT sends "Interval LVT Request Messages" to all other nodes. It then goes back to regular processing.
2. Each node on receiving a "Interval LVT Request Message":
   - Finishes distributing all messages from the communication buffer.
   - Calculates the minimum of the current lowest event time of all its entities and the time of the earliest message sent during this interval. A new interval is then started.
   - Sends a "Interval LVT" message with the calculated value back to the GVT calculating thread.
3. On receiving the last "Interval LVT Message" from the other threads, the GVT calculating thread calculates the GVT as the minimum of all received "Interval LVT"'s and its own "Interval LVT" (calculated as in (2).

4. The GVT calculating thread sends a "GVT notification" message with the new GVT value to the other threads and performs its own deadwood collection processing based on the new GVT.

5. On receiving a new GVT the threads perform deadwood collection to clean up its memory.

Simple Distributed GVT algorithm

The simple distributed GVT algorithm works as follows: Each time a message is sent or received its sequence number and message time is noted by the subsystem. Periodically the subsystem is instructed to send a summary of this information to the central GVT controller. The information sent is the sequence number interval for each output channel and the lowest message times for each interval. Also the last sequence number received on each input channel is also sent and the Local Virtual Time for the node. The GVT controller looks at this information and can determine a lower bound on the time messages that are still "in flight" and must be used in the GVT calculation. Periodically the GVT controller calculates the GVT and sends it to all other nodes. This calculation is simple: the sent sequence number intervals for each channel are remembered (a list), when a new value for the last received sequence number for that channel is received, the intervals before that sequence number are deleted and the minimum time of the remaining intervals is recalculated. This protocol uses O(N) messages of size O(N) per periodic GVT calculation phase.

4.6 Algorithm-Specific COMPOSE Constructs

4.6.1 Sequential Algorithms

If the simulation is executed with one virtual processor, the runtime will default to sequential execution using the global event list algorithm based on splay trees. No optimizations are needed to execute the model in this mode, which is the default mode for COMPOSE models.

4.6.2 Null Message Algorithm

To execute in this mode the "Starter Entity" must execute the "DoNullMessageSyncronization()" method.

An entity notify the runtime that it will execute conservatively by calling:

- **IAmConservative();**

With the Null Message algorithm each entity must notify the system of its local topology, i.e., which entities it will send to. The following methods allow an entity to add or delete destination entities from its local topology.

- **AddSuccessorEntity(const EntityNameType& EntityName);**
- **DeleteSuccessorEntity(const EntityNameType& EntityName);**

The entity must also specify the lookahead:

- **void SetLookahead(LookaheadTime, EOT_Ceiling);**

The "LookaheadTime" (of type TimeType) is the minimum delay that on receiving a message the entity will send a message to another entity. The "LookaheadTime" is used to calculate the "Earliest Output Time" which is defined to be the earliest time that the entity could possibly send a message. The "EOT_Ceiling" (TimeType) is an optional parameter that specifies a ceiling on the EOT and is normally used in conjunction with "Timeout Events".

- **void SetLookahead(DestinationEntityName, LookaheadTime, EOT_Ceiling);**

This version of "SetLookahead" allows the user to specify lookahead times for each destination entity. This gives finer control over EOT's than the global lookahead approach. The two "SetLookahead" call types are mutually exclusive and calling both in the same entity will produce a runtime error.

If the topology or lookahead are incorrectly specified, the simulation will terminate with a runtime error.

Dynamic creation of entities with the conservative simulation algorithm is very tricky. Suppose entity "A" wants to create entity "N" that will have a reference to existing entity "B", then there must be a "positive lookahead path" from entity "A" to entity "B" with total lookahead time "L". Furthermore, the new entity "N" must not send a message to "B" before delaying by at least "L" (otherwise the previous lookahead guarantees would be violated).

4.6.3 Optimistic Algorithm

With Optimistic execution, the entity's state must be saved and restored. COMPOSE saves the memory image of the entity's member variables. Changing dynamic memory pointed to by member variables must also be saved. To accomplish this, all such dynamic memory must be allocated from a state-saved memory allocator as follows:

```
DataPtr = new(StateSavedMemory) DataType(...);
```

The runtime system will keep track of this memory and restore it to its proper state when a rollback occurs.

Non-changing data structures can be allocated with the standard "new" operator.

Note that if even one bit of changing state is not under the control of the optimistic runtime, then the simulation will fail to work correctly as the entity's state will not be rolled-back properly. This mistake is extremely easy to make, as it will occur with any use of global variables or memory allocated through the standard "new" call supported by C++.

Optimistic Entity Termination and Abortion

With a dynamic topology, events that create other entities can be rolled-back which causes the creations to be rolled-back. Thus, entities can be created and then rolledback out of existence (aborted). This causes an ambiguity when the entity's destructor is called as it is not known whether the entity terminated naturally or was aborted.

To differentiate between the two cases the following method is provided:

- **Bool ICompletedNormally();**

This method will return true when the entity completed normally and false when it is aborted (rolled-back out of existence). The simulation programmer should dump results only if the entity completed normally.

Optimizations

- **SetStateSavingInterval(int Interval);**

The "SetStateSavingInterval" method allows the entity to change its state saving interval. State saving interval defaults to 1 (saved every event). Upping this value lowers state saving costs and memory requirements, but increases the cost of doing a rollback.

- **TurnOnLazyCancellation();**

This method switches to "lazy cancellation" of anti-messages, a variant of the optimistic protocol. The default is aggressive cancellation that sends anti-messages immediately. Lazy cancellation delays the sending of anti-messages in the hope that the re-execution of the entity will produce the exact same messages that were produced before. When this occurs, the anti-messages and the duplicate messages do not have to be sent.

5. Database Models

The primary focus of the Phase I effort was to implement a prototype parallel simulator using C++ and evaluate the feasibility of simulating very large parallel databases accessing many terabytes of data running on thousands of processors. In modeling parallel databases, there are two important considerations: the hardware architecture and the data model.

5.1 Hardware Architectures

Three types of architectures have commonly been used for parallel databases:

- Shared-memory architectures: All processors share all memory and all disk resources available within the system.
- Shared-disk architectures: Each processor has private memory but all disks are shared among all processors
- Shared nothing, or clustered, architectures: Each processor can access a unique set of memory and disk elements. Access to the remote disk or memory element must be sent as a message using the interconnection network.

In recent years, almost all parallel databases, particularly in the commercial context, have migrated towards the shared-nothing architecture. Two primary reasons have led to this development: first, this architecture is able to exploit advances in processor, memory and disk technologies directly. Second, the cluster computing architecture is scalable, both in terms of the maximum number of devices that can be connected together without significant increase in the interference in the communication network, as well as in its ability to do this in an incremental manner. Most commercial vendors are beginning to exploit this architecture for their parallel databases; Teradata was among the first to use this architecture for their parallel databases. The focus of this effort was on shared nothing architectures.

5.2 Parallelism in Relational Databases

In relational databases, parallelism may be exploited via inter-query parallelism (multiple OLTP queries can be processed simultaneously by multiple processors) or intra-query parallelism where a complex query is decomposed into multiple steps, which may be executed in parallel. Scans and joins are the most common operators in complex queries and their execution time has the most impact on overall database performance. A relational query can be executed as a dataflow graph which can exploit both pipelined and partitioned parallelism. In general, pipelined parallelism offers limited opportunities for speedup because a query has only a few steps. In contrast, partitioned parallelism offers significant opportunities for scaling with processors, disks, and perhaps most importantly, the size of the database. Given the significantly larger potential for superior performance with partitioned parallelism, this effort was devoted primarily to the evaluation of workloads amenable to this form of parallelism.

5.3 Data Partitioning

Parallel databases partition a relation to improve performance by exploiting the collective bandwidth from accessing multiple disks in parallel. This involves distributing the tuples in a relation over multiple disks. Three primary partitioning strategies have been used: *round robin* that simply allocates the tuples among the disks in a round robin fashion; *range partitioning* that clusters tuples based on their proximity with respect to specific attributes; and *hash partitioning* which use a hashing function on a specified attribute of the tuple to decide its placement on a specific disk. The latter two techniques are used commonly: for instance Teradata uses hash partitioning whereas both Tandem and Oracle have used range partitioning. Both techniques have advantages in specific applications: range partitioning clusters related data together, but tends to be application-specific. So if a given relation is used in many different types of applications, the clustering may lead to sub-optimal performance. It is also the case, that queries with even low amounts of selectivity can cause severe load imbalance. In contrast, hashing tends to randomize data rather than cluster it and is more suitable for sequential and associative scans (where data with a given value of an attribute are accesses together). Some databases use a variation of range partitioning where the range is not selected uniformly; rather the data is distributed based on the access frequency of tuples.

Partitioning raises a number of performance issues. Note that although partitioning generally tends to improve performance, beyond a limit it can cause unnecessary fragmentation of the relation leading to an increase in the execution time [Ghandeharizadeh and DeWitt 1990]. In Phase I, our aim was to evaluate performance of workloads and datasets using the hash partitioning that has been used by Teradata.

5.4 Prototype Database Models

The prototype model simulates a parallel database running on a parallel architecture. The hardware architecture can be viewed as a set of N nodes linked by a multi-path interconnection network such that no memory or other resources within a node is directly shared or accessed by another node. Each node is a symmetric multiprocessor and consists of P processors, C disk controllers and D disks, where controller c_i controls d_i disks. The processors are connected to the disk controllers via a common bus. Memory elements are not shown in this schematic because memory performance is critical in data base models. A simple schematic of the hardware architecture is shown in Figure 2 below.

From a software perspective, each node has two types of threads: PE threads and AMP threads that together share the processors available within each node. Each PE thread controls a unique job, that consists of a sequence of steps. Some steps are required to be executed on all the nodes (for instance for a query that does a massive scan) and others may only be executed only on a specific node. Each step is executed by creating an AMP thread that corresponds to the operations to be performed on a subset of the disks belonging to that node. Each AMP thread created by a step owns a unique set of disks, such that a disk d_i may only be accessed by some AMP thread, say amp_j. The AMP thread is the primary simulation object in the model and it executes the following two tasks to simulate the execution of a query on its subset of disks: first, is the *local* step, where it performs the local computation and the IO operations necessary to compute a local result and second, is the *transmit* step, where it transmits the result to the PE that owns this step.

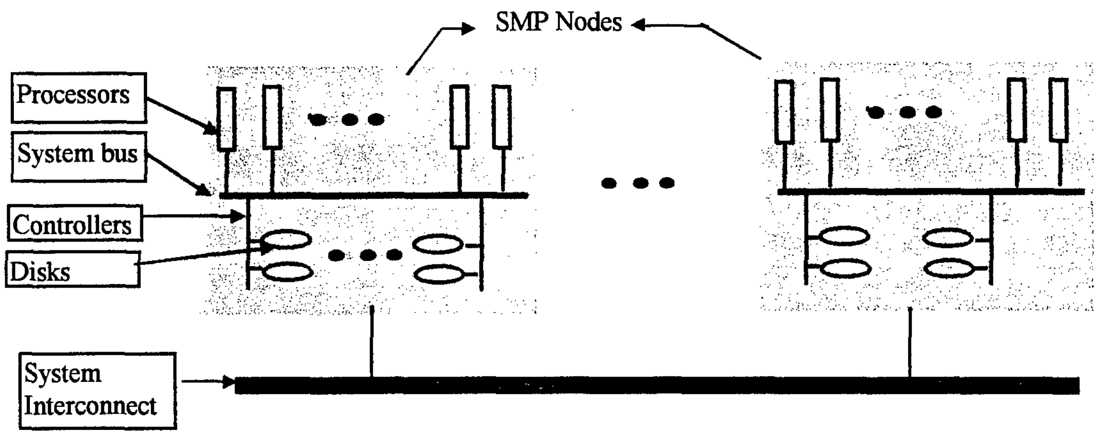

Figure 2: Schematic of A Parallel Database

The first task is performed by implementing it as a loop that *uses* the following facilities in sequence: CPU, bus interconnect internal to the node, disk controller(s) that control its unique disk subset, and a set of disks. The number of loop iterations and the amount of time used by each of the preceding components per iteration is parameterized. The loop is used to simulate the resource sharing and contention among the different threads within the SMP node as each AMP thread performs the required read/write operations on the disks, performs necessary computations, and possibly moves data between the multiple nodes.

The second task simulates transmission of the results computed by the AMP thread to the PE as a sequence of one or more *return packets*. This task uses the CPU component within the node that performed the computation, utilizes some time on the global interconnect, and uses CPU at the destination node that contains the PE thread that spawned this step. Once the preceding tasks have completed, the AMP thread terminates.

A *scan* operation is simulated by performing a large number of *local* steps in a loop. A hash join or other operation that involves a redistribution of data requires transmitting the result of a query back to the corresponding job thread. This is simulated by inserting appropriate *transmit* steps into the preceding scan loop.

In the simple model, each of the preceding components except the interconnect (CPU, system bus, disk controller, disks) is modeled by a FIFO server. As each node has an 8 processor CPU, the processors are modeled by a service station with 8 FIFO servers. The model computes the average waiting time and utilization for each server.

A number of communication models were implemented to simulate the system interconnect. In the initial version, the interconnect was modeled as a simple global resource and simulated using a common FIFO queue. However as this did not appear to be an appropriate representation of the system (it introduced large queuing delays even among messages that were being transmitted between independent source-destination pairs), the interconnect was subsequently modeled by a distributed direct link based model. The resulting model was used for the initial comparison between PARSEC, CSIM, and COMPOSE and is henceforth referred to as *Model 1*. In the direct link model, there are no shared interconnect path ways and contention is only modeled on links into or out from a node. In Model 1, query result was simulated by sending messages to the proper "job thread", but data redistributions only simulated the use of the communication links. Note, a data redistribution will send to all other nodes, while a query result communication will only send data to the node that is executing the corresponding "job thread". The next evolution of this model (*Model 2*) was to improve the fidelity of the data redistribution queries in two ways:

- The *transmit* phase occurs periodically during the *local* loop, rather than only at the end
- Actual messages are sent to the destination node(s) to simulate the use of the interconnect on data redistributions, rather than simply simulating these as delays at the local interconnect facility. This allows an interconnect communication model with shared pathways to be modeled by the system. For example, congested shared interconnect pathways will increase the communication time and thus may become a bottleneck and thus increase the time to complete a database step.

Both Model 1 and Model 2 have a very low computation granularity because each system component is modeled by a server where the only computation in response to an incoming message is the computation of a service time for the corresponding step. Even the time to execute a read or write request for a data block is modeled simply by delaying the corresponding AMP thread by an interval that is proportional to the length of the data block that is being read or written. No algorithmic or data distribution properties of the system are simulated. Given the extremely low computation granularity, it did not appear reasonable to model each facility as an entity as this would severely degrade the sequential performance. Instead, each facility was modeled by a 'variable' within a single entity that simulated each node of the target hardware. This approach was dubbed the 'Variable Facilities' approach as each server (or facility) is modeled by a single variable.

Using the 'Variable Facilities' approach, the sequential COMPOSE implementation of Model 1 was found to be 3X faster than the functionally equivalent CSIM model. This exercise also served to illustrate the classic tradeoff between programming effort and ease of use versus raw execution speed. Clearly the use of variable facilities considerably improved the performance of the sequential simulator but the performance benefits came at the expense of considerable programming efforts when compared with the ease of implementing a model where each facility was programmed as a simulation object.

5.5 Model Parallelization

Large data mining database operations have considerable internal parallelism which has prompted the development of parallel databases on massively parallel architectures. It follows that a simulation model of such a system must also have considerable potential to benefit from parallel execution. However, there is one significant difference between the physical database and its model: the physical database has a large computation granularity because each node may include a large set of relations. In contrast, as the model simulates the local processing simply by estimating a delay the computation granularity for the model is much smaller. The small computation granularity may, in some cases, be compounded by frequent communications among the nodes that are mapped to different processors of the simulator. These messages can create event chains with small delays weaving back and forth between the processors in the simulator. These chains will force the critical path of the parallel simulator to become too dependent on the message passing latency of the simulating machine and thus likely to kill the possibility of parallel speedups. This is why the communication between nodes in the simulator is of primary importance to parallel simulation. With the conservative parallel simulation algorithm, even the possibility of such chains will cause a problem, as this algorithm must have guarantees that include the worst case. This is not the case with the optimistic algorithm as it can tolerate such event chains if they are infrequent. Fortunately, such chains do not occur very often with the data mining workloads, which are characterized by long running scans and joins.

The models described in the previous section were executed using both conservative and optimistic simulation algorithms and the results of the experiments are presented next.

6. Results

The Phase I study demonstrated the feasibility of using parallel simulation with COMPOSE to achieve scalability for database modeling. As summarized in this section, we were successful in developing a prototype implementation of an object-oriented parallel simulation environment (COMPOSE), develop a simplified model of a Teradata parallel database, simulate simple queries that were similar to the queries from the TPC-D benchmark, validate the model with respect to a model developed by NCR using CSIM, a commercial C-based simulation library, demonstrate the functional equivalence between a COMPOSE and CSIM model of a Teradata database, and most importantly demonstrate the considerably superior performance that could be obtained with the COMPOSE simulator. Two different types of workloads were used to respectively represent the best and worst case scenarios: the first workload represented a large scan of the database that can effectively exploit the partitioned parallelism. Due to the large amount of inherent parallelism, the parallel implementation of the simulator was expected to produce good speedup. The second workload represents a query which includes steps that requires significant data redistribution. Such steps would occur, for example, during a hash join that must redistribute intermediate tables produced to evaluate the original query. Because of the frequent communications, this type of query has relatively poor performance on the physical database and even the parallel performance of the simulator was expected to be relatively poor. This section presents the primary results from the study.

6.1 Model 1: Sequential Performance

A model simulated a database running an 8-nodes with 8 jobs distributed to a corresponding number of front-end (PE) threads. The first step in the study was to validate the output from the COMPOSE model against the CSIM model developed by NCR personnel. A detailed trace was produced to ensure that the *event sequences were identical* in both cases. The execution time of the sequential implementations were compared next. As seen from Figure 3, the sequential performance of the COMPOSE model was almost 3 times better than the CSIM model. The 1-node implementation of the conservative protocol performed even better than the sequential protocol, due to its lower context switching overheads.

Simulator	**Time(secs)**	**Speedup**
CSIM	237	--
COMPOSE Sequential	82	2.9
COMPOSE Conservative	75	3.1

Figure 3 Comparative Simulator Performance on a Single Node of the Sparc 1000

6.2 Model 1: Conservative Performance

6.2.1 Parallel Model Refinement

In the conservative method, the simulation programmer must provide the simulation system with "lookahead" information for each simulation object. The lookahead represents a guarantee by the entity about the earliest future time at which it will send a message; e.g., at time 10 a simulation entity can declare it will not send a message until time 15 which implies that it has a lookahead of 5. In general, the larger the lookaheads the better is he parallel performance.

The lookaheads for every "Job Thread" and "Worker Thread" were calculated and the lookaheads were aggregated for each node. One of the methods used was to calculate a total "contention-free" delay (lookahead) for the thread and then at each event subtract the "contention-free" delay for that event. This works well with read scans as the delay before they will send a completion message to the "Job Thread" is quite long. Since the earliest output time to each node may be different, it makes sense to specify the "lookaheads" for each node rather than use a single global value. With data distributions or when results are returned to the "Job Thread", there is also an "contention-free" delay that can be utilized. To maximize this delay, the lookaheads must be based on the specific model's exact communication pattern. A good characteristic of this model is that inter-node communication on large data mining queries is usually a steady stream of packets. The conservative algorithm works best with predictable steady stream communication and does not work well with infrequent and unpredictable communication with small delays. A low delay "killer" query would be a simple retrieval of a single data record as follows:

- A Job thread is created and sends a message to another node to get some data.
- The node has the data in a cache and can ship it back, with negligible delay, to the Job Thread.

It may be necessary to modify the model to better exploit lookahead guarantees. For example, it is known exactly when a transaction will enter the system and thus the transaction can be inserted before its execution and used in lookahead calculations. Another problem is the transition from one job step to the next. At the end of a step, all "Worker Threads" must report that they are done. If the work done at the end of the step is very small, there will be very little delay between the last message and the start of the new step. To increase lookahead on long running steps, it may be necessary for the "Worker Threads" to send estimates for which the "Job Thread" to calculate an estimate for the start of the new step.

As the previous examples demonstrate, calculating lookaheads is a difficult task and depends on the specific model, model parameters and the workloads to be modeled. There will be certain combinations of parameter and workloads that will have poor lookaheads and thus will result in poor parallel performance with the conservative algorithm. The optimistic parallel simulation algorithm is probably more appropriate in these situations. If the interconnect delay is large enough, then it may not be necessary to bother with the all the above lookahead calculations as it would sufficient to just rely on the lookahead of the interconnect delay.

6.2.2 Conservative Performance

The conservative COMPOSE model was migrated to two parallel architectures – a SUN Sparc1000 architecture with 8 symmetric multiprocessors and a 4-processor SMP platform from DELL running Windows NT, where each processors is a Pentium Pro. We first consider the performance on the Sparc. As expected, the parallel implementation yields excellent speedup for the scan query (Figure 4). In the figure, the column titled speedup presents the performance improvement as compared with the sequential implementation and the column titled scaled speedup compares the parallel performance with respect to the sequential CSIM model. Figure 5 presents the performance for the hash-join query with data redistribution. Although the large amount of communication with limited lookahead causes the speedup to become worse, parallel execution still results in some benefit: with 4 processors there is a factor of 2 improvement in the performance. Further, the simulation model assumed the very worst case, where the communication latency on the target hardware was assumed to be 1 microsecond! For realistic architectures, the communication latency would be considerably larger, and this effect was investigated on the DELL platform.

Two sets of experiments were run on the DELL. The first set was similar to the experiments run on the Sparc 1000; the purpose was to verify that the superior parallel performance could be replicated on an architecture with state of the art sequential processors. As seen in Figure 7, the speedup obtained for both the scan and the join queries were comparable to those obtained on the Sparc1000 with 4 processors. For the second set of experiment, the communication latency of the target architecture was varied. As the latency is increased, the lookahead in the model increases proportionately which should lead to better parallel performance for the model. As seen from the graph in Figure 8, this was indeed found to be the case, with the speedup for the join query approaching that of the scan for communication latencies of over 500 microseconds. Although, contemporary parallel architectures have a communication latency of between 20-100 microseconds, for the large amounts of data (typically in the Megabytes) that are typically transmitted for database queries, communication delays of hundreds of microseconds and even milliseconds are not unusual. For the Sparc1000 case, when a communication latency of 1 ms was introduced, the speedup went back to that obtained for the scan queries (Figure 6). The evidence of significant potential for parallel simulation to improve model execution time for this application is unmistakable!

No. of Processors	Execution Time(s)	Speedup	Scaled Speedup
1	285	1	3.1
2	151	1.88	5.8
4	83	3.4	10.6
8	50	5.7	17.6

Figure 4 Parallel Performance of a Scan query: Sparc 1000

No. of Processors	Execution Time(s)	Speedup	Scaled Speedup
1	204	1	3.1
2	152	1.3	4
4	102	2	6
8	89	2.3	7.1

Figure 5 Parallel Performance of a Query Involving Data Redistribution: Sparc 1000

No. of Processors	Execution Time(s)	Speedup	Scaled Speedup
1	195	1	3.1
2	107	1.8	5.6
4	55	3.5	10.8
8	35	5.5	17

Figure 6: Data Redistribution Query with Delay Model: Sparc 1000

No. of Processors	Scan		Join with redist.	
	Exec time	speedup	Exec time	speedup
1	85	1	53	1
2	47	1.8	39	1.4
4	25	3.4	28	1.9

Figure 7: Parallel Performance on 4-way SMP with Intel Pentiums and Windows NT

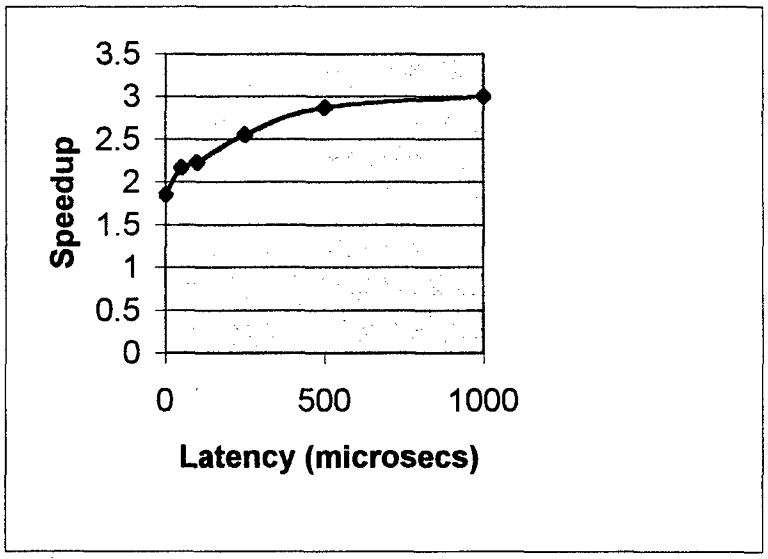

Figure 8: Speedup variation as a function of Communication Latency of target hardware

6.3 Model 2: Conservative Performance

As discussed in the previous section, the primary difference in Model 2 is that the communication that arises in the physical system when simulating a scan query is

modeled more accurately. The experiments were repeated for the heavy communication scenario for Model 1 as this was the most challenging workload for Model 1 in terms of delivering appropriate parallel performance. The speedup for both the Sparc1000 and the Dell machines is presented in Figure 9 as a function of the number of processors.
As communication among the different entities in the model increases, a degradation in the parallel performance of the model was expected because the lookaheads tend to become worse than in the case where all communication occurs only at the end of the step. This can be seen clearly from noticing that the 1-node conservative model now becomes substantially less efficient than the 1-node sequential model. Note also that the sequential execution time on the DELL platform is almost half that of the Sparc1000, but both architectures demonstrate he benefits of parallel model execution. It was interesting that even for this configuration, reasonable benefits could be obtained from parallel execution on both platforms!

Processors	**Sparc 1000**		**Dell 4-Way**	
	Time	**Speedup**	**Time**	**Speedup**
1; sequential	69	1	33.1	1
1; conservative	81	0.85	38.0	0.87
2	46	1.5	22.2	1.5
4	25	2.8	12.5	2.65
8	15	4.6	---	---

Figure 9: Parallel Performance of Model 2

6.4 Model 2: Optimistic Performance

A number of alternative model implementations were also used to evaluate the performance of the optimistic synchronization algorithms for this model. Recall that in the very first implementation, every thread and facility were represented by a separate entity. Although this model was easy to develop it had terrible sequential performance and, interestingly, the optimistic performance was also poor. The implementation of Model 1 using the "variable facilities" concept greatly improved the basic efficiency of the sequential model implementation but it also complicated the design and development of the optimistic model. This is because the "variable facility" model merged all objects that belong to a single *node* of the target architecture into a single simulation entity. The default state saving technique in the system was the copy state saving and this introduced substantial overheads for optimistic execution of Model 1. As expected, experiments showed that running the model naively in optimistic simulation mode achieved extremely poor performance (in some cases, 7X slower on 4 CPUs than the sequential version). The reason for this is that copy state saving was too expensive. In these experiments, periodic state saving helped to reduce this overhead, but increasing the period increased the overhead of re-executing events during "coast forward" processing. Obviously, a more sophisticated method of state saving was necessary. In addition to state saving overheads, the costs of GVT computation also had an impact on the performance. Figure 10 shows a sample scenario that demonstrates the impact of varying these parameters on the performance of the parallel optimistic model. The experiments were executed on the DELL platform. The second column indicates the number of events executed by an entity between successive state saving operations and column 3 indicates the number of times GVT was computed/second of physical time. Increasing the frequency of GVT computations improves performance because it reduces the memory requirements of the model. Clearly the state saving problem had to be addressed in a different way if the optimistic techniques were to be useful.

No. processors	State Saving Interval	GVT calculations/sec	Exec. Time (secs)
1; sequential	---	---	8.9
4	10	20	80
4	100	20	12.9
4	100	100	11.9
4	200	20	11.2
4	200	100	9.9

Figure 10: Impact of state saving and GVT computation frequency on parallel performance

The method used for this report is a incremental state saving technique that only tracks the part of the entity state that is actually modified by the execution of an event. For each event, the entity prepares a changes record that records the part of the entity state that was actually modified and this "change" record is then inserted into a queue. During a rollback, the changes for the event can be undone by restoring the state of the corresponding simulation object. Previous states can be restored by undoing all events in the queue in backwards order, in the same way a undo function works in a word processor or text editor. To prevent overflowing the computers memory, the front of the "change" record queue needs to be garbage collected as soon as the simulator determines that the past events will never be rolled back. COMPOSE provides a simple interface to allow a model to be notified of rollbacks and garbage collections.

The implementation with the optimistic algorithms with incremental state saving was used to evaluate the performance of the configuration that included the very heavy communication scenario which had previously yielded he dramatic slowdowns described previously. As seen from Figure 11, using incremental state saving did reduce the state saving overheads substantially enough such that it was now possible to get improved performance with parallel model execution on the Sparc 1000:

Processors	Exec. Time (secs)	Speedup
1	66	1
4	47	1.4
8	31	2.1

Figure 11: Optimistic Performance of Model 2

Though greatly reduced, state saving and keeping track of past events in the optimistic runtime are the the major overheads with these results. If the model with this test case is executed on 1 CPU but states are saved and managed the model is slowed down by a factor of 2.6X. Eliminating the incremental state saving in the model lowered this to about a factor of 2 slower which represents the COMPOSE optimistic mode overhead. One inefficiency with this setup is that because we used a general optimistic simulator both the model and COMPOSE keep and manage event/state queues. A specific streamlined simulator for this model would be faster at the cost of not leveraging the general simulator. Undoubtably, both the overheads in COMPOSE and the modeled incremental state saving can be improved to lower this overhead somewhat. In the test case, the overhead is overpowered by throwing processors at the problem. Given a larger model configuration, a parallel machine with more CPUs and good memory bandwidth should be able get better speedups with these results. The 8 node test case may be limited by the small shared memory bandwidth of the Sparc 1000.

7. Conclusion

The Phase I study demonstrated the feasibility of using parallel simulation to achieve scalability for database modeling.

As summarized in this report, during Phase I

- we designed a performance evaluation capability for scalable systems,
- developed a prototype implementation of an object-oriented parallel simulation environment (COMPOSE),
- developed a simplified model of a Teradata parallel database,
- simulated simple queries that were similar to the queries from the TPC-D benchmark,
- validated the model with respect to a model developed using CSIM, a commercial C-based simulation library, and demonstrated the functional equivalence between a COMPOSE and CSIM model of a parallel database, and
- demonstrated the considerably superior performance that could be obtained with the COMPOSE simulator for different workloads.

Two different types of workloads were used to respectively represent the best and worst case scenarios: the first workload represented a large scan of the database that can effectively exploit the partitioned parallelism. Due to the large amount of inherent parallelism, the parallel implementation of the simulator was expected to produce good speedup. The second workload represents a query which includes steps that requires significant data redistribution. Such steps would occur, for example, during a hash join that must redistribute intermediate tables produced to evaluate the original query. Because of the frequent communications, this type of query has relatively poor performance on the physical database and even the parallel performance of the simulator was expected to be relatively poor.

A model simulated a database running an 8-nodes with 8 jobs distributed to a corresponding number of front-end (PE) threads. As described in this report, the sequential performance of the COMPOSE model was almost 3 times better than the CSIM model and the 1-node implementation of the conservative protocol performed even better, due to its lower context-switching overheads. The COMPOSE model was subsequently migrated to two parallel architectures – a SUN Sparc1000 architecture with 8 symmetric multiprocessors and a 4-processor SMP platform from DELL running Windows NT, where each processors is a Pentium Pro. As expected, the parallel implementation yields excellent speedup for the scan query on both platforms, with the Sparc1000 yielding a speedup factor of 17 on 8 processors as compared with the sequential CSIM model and the 4-processor PC yielding a speedup of almost 10 as compared with the sequential CSIM model. For the hash-join query with data redistribution, although the large amount of communication with limited lookahead causes the speedup to become worse, parallel execution still results in some benefit: with 4 processors there is a factor of 2 improvement in the performance. Further, the simulation model assumed the very worst case, where the communication latency on the target hardware was assumed to be 1 microsecond! For realistic architectures, the communication latency would be considerably larger. in our study, this was indeed found to be the case, with the speedup for the join query approaching that of the scan for communication latencies of over 500 microseconds. Although, contemporary parallel architectures have a communication latency of between 20-100 microseconds, for the large amounts of data (typically in the Megabytes) that are typically transmitted for database queries, communication delays of hundreds of microseconds and even milliseconds are not unusual. The evidence of significant potential for parallel simulation to improve model execution time for this application is unmistakable!

Based on the positive results obtained from the Phase I study, we have initiated development of a state of the art performance modeling tool for scalable data base systems using the COMPOSE parallel object-oriented simulator. A proposal for Phase II funding for this purpose has been submitted to the DARPA ITO office.

8. Bibliography

M. Abbott and L. Peterson. A language-based approach to protocol implementation. Technical Report Tech. Rep. 92-2, University of Arizona CSD, July 1992.

A. Allen, Introduction to Computer Performance Analysis with Mathematics, Academic Press, Boston, 1994

Wing Au and Rakesh Jha. C3I Parallel benchmark suite: Map-image correlation. Technical Report, Honeywell Technology Center, April 1997.

D. Baezner, G. Lomow, and B. Unger, "Sim++: The transition to distributed simulation," *Proceedings of the 1990 SCS Multiconference on Distributed Simulation,* San Diego, California, January 1990, pp. 211-218.

R. Bagrodia, K. Chandy, and W-T. Liao, A unifying framework for distributed simulations. *ACM Transactions on Modeling and Computer Simulation,* Vol. 1(4), October 1991, pp. 348-385.

Rajive Bagrodia, Stephen Docy, and Andy Kahn ,Parallel Simulation of Parallel File Systems and I/O Programs, SuperComputing '97 — SC97, November 15-21, 1997, San Jose, CA.

R. Bagrodia and W. Liao, "Maisie: A language for the design of efficient discrete-event simulations," *IEEE Transactions on Software Engineering,* Vol. 20(4), April 1994, pp. 225-238.

R. Bagrodia, R. Meyer, B. Park, H. Song, Y. Chen, X. Zeng, J. Martin, M. Takai; "PARSEC: A parallel simulation environment for complex systems," *IEEE Computer Magazine,* 1998 (to appear).

R. Bagrodia and C.-C. Shen "MIDAS: Integrated Design and Performance Evaluation of Distributed Systems,"; IEEE Transactions on Software Engineering, October 1991, Vol. 7(10), pp. 1042-1058.

Carrie Ballinger. Relevance of the TPC-D benchmark queries. Technical report, NCR Parallel Systems, 1996.

B. Bayerdorffer, "Distributed Programming with Associative Broadcast", HICSS , Jan. 1995.

G. Booch, J. Rumbaugh and I. Jacobson "Unified Modeling Language User Guide" (Addison-Wesley, Englewood Cliffs, 1997)

H.~Boral, W.~Alexander, and L.~et~al Clay. Prototyping Bubba, a highly parallel database system. IEEE Transactions on Knowledge and Data Engineering, 2(1):4--24, March 1990.

E.~A. Brewer, C.~N. Dellarocas, A.~Colbrook, and W.~E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator}. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, Cambridge, MA 02139, 1991.

J. Briner, J. Elis, and G. Kedem, "Breaking the barrier of parallel simulation of digital systems," Proceedings of ACM/IEEE Design Automation Conference, 1991.

Y. Chen and R. Bagrodia, "Shared Memory Implementation of A Parallel Switch-level Simulator," Proceedings of the 12th Workshop on Parallel and Distributed Simulations, Banff, Alberta, Canada, May 1998.

Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. ACM Transactions on Computer Systems</em>, 14(3):225--264, August 1996.

Culler, D., R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. VonEiken, "LogP: Towards a Realistic Model of Parallel Computation", Proc. 4th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming (PpoPP '93),San Diego, May 1993, pp. 1-12.

H. Davis, S. R. Goldschmidt, and J. Hennessey. Multiprocessor simulation and tracing using Tango, Proceedings of the 1991 International Conference on Parallel Processing (ICPP'91)}, pages II99--II107, August 1991.

S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, "GTW: A time warp system for shared memory multiprocessors," 1994 Winter Simulation Conference, Lake Buena Vista, FL, December 1994.

Deelman, E. et al, POEMS: End-to-end Performance Design of Large Parallel Adaptive Computational Systems, Workshop on Software Tools & Performance, 1998 (submitted)

Defense Modeling and Simulation Office, HLA Interface Specification 1.1, Feb 1997.

Dewitt and J. Gray, Parallel Database Systems: The Future of High Performance Database Systems, Communications of ACM, 35(6), June 1992, pp85-98.

D.~Dewitt, S.Ghandeharizadeh, D. Schneider et al, The gamma database machine project IEEE Transactions on Knowledge and Data Engineering}, 2(1):44--63, March 1990.

P. Dickens, P. Heidelberger, and D. Nicol, "Parallelized direct execution simulation of message-passing parallel programs," IEEE Transactions on Parallel and Distributed Systems, Vol. 7(10), October 1996.

Ghandeharizadeh, S. and DeWitt D.J., Performance Analysis of Alternative Declustering Strategies, Proc. Sixth International Conference on Data Engineering, Feb, 1990

N. Gunther, The Practical Performance Analyst, McGraw Hill, New York, 1998

IEEE Transactions on Knowledge and Data Engineering, 2(1), March, 1990

R. Jain, The Art of Computer Systems Performance Analysis, John Wiley & Sons, New York, 1991

D. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7(3), July 1985, pp. 404-425.

V. Jha and R. Bagrodia, "A performance evaluation methodology for parallel simulation protocols," *Proceedings of the 10th Workshop on Parallel and Distributed Simulations*, Philadelphia, PA, May 1996, pp.180-183.

J. Martin and R. Bagrodia "COMPOSE: An object-oriented environment for parallel discrete-event simulations," *Proceedings of the 1995 Winter Simulation Conference*, December 1995, pp. 763-767.

S. McCanne, S. Floyd et al " NS Manual page , http://www-nrg.ee.lbl.gov/ns/man.html

Menlo Software, DBAware 2.0 User Manual, http://menlosoftware.com/

Barton P. Miller, et al .The paradyn parallel performance measurement tools .IEEE Computer, 28(11), November 1995.

J. Misra, "Distributed discrete-event simulation," *ACM Computing Surveys*, Vol. 18(1), March 1986, pp. 39-65.

MPI Forum. MPI: A message passing interface. Proceedings of 1993 Supercomputing Conference, Portland, Washington, November 1993.

V.A.Norton, Frederica Darema, and G.F. Pfister. Using a single-program- multiple-data computational model for parallel execution of scientific applications. IBM Research Report, RC 11552, IBM T.J. Watson Research Center, Yorktown Heights, New York, November 1985

S. Prakash, *Performance Prediction of Parallel Programs*, Ph.D. Dissertation, UCLA Computer Science Department; July 1996

Precise Software Solutions, Precise/SQL Manual, http://www.precisesoft.com/

J. Rasure, D. Argiro, T. Sauer, and C. Williams, A Visual Language and Software Development Environment for Image Processing," International Journal of Imaging Systems and Technology, Vol. 2, pp 183-199 (1990).

Daniel A. Reed, et al, Pablo: An extensible performance analysis environment for parallel systems Pablo Research Group Technical Report; http://bugle.cs.uiuc.edu/Pablo.html, 1992.

S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood, "The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers," Proceedings of the 1993 ACM Sigmetrics Conference, May 1993, pp. 509-518.

J. Rumbaugh, et.al. Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ, 1991.

Rosenblum, M., S.A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The SimOS approach," IEEE Parallel and Distributed Technology, Winter 1995, pp. 34-43.

H. Schwetman, "CSIM: A C-based process oriented simulation language," Proceedings of the 1986 Winter Simulation Conference, 1986, pp. 387-396.

S. Shlaer and S. Mellor "Object Lifecycles: Modeling the World in States" Yourdon Press, New York, 1992

J.~Short, R.~Bagrodia, and L.~Kleinrock. Mobile Wireless Network System Simulation. Proceedings of the 1995 ACM International Conference on Mobile Computing and Networking, Berkeley, CA, November 1995.

W. Stallings, SNMP, SNMPv2 and RMON Practical Network Management, Addison Wesley, 1996

NCR Corp, Teradata Capacity Planner White Paper, 1997.

UCLA Parallel Computing Laboratory, *PARSEC User Manual*, Release 1.0, Computer Science Department, University of California, Los Angeles, CA, http://pcl.cs.ucla.edu, February 1998.

X. Zeng, R. Bagrodia and M. Gerla, "GloMoSim: A library for the parallel simulation of large wireless networks," Proceedings of the 12th Workshop on Parallel and Distributed Simulations, Banff, Alberta, Canada, May 1998.